

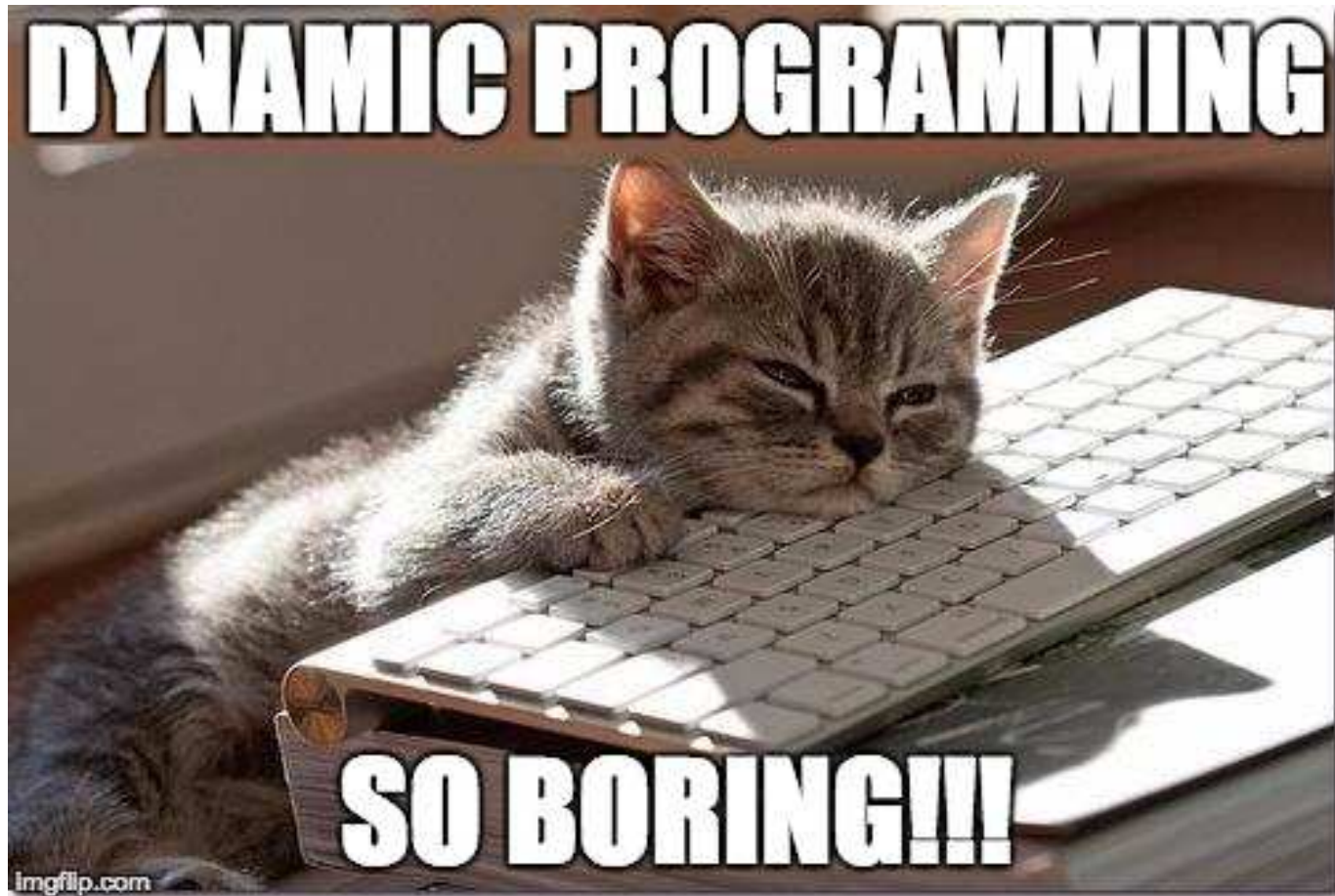
Lecture 14

Greedy algorithms!

Announcements

- HW6 Due Friday!
 - ***TONS OF PRACTICE ON DYNAMIC PROGRAMMING***
- Sometimes I have **hidden slides** in PowerPoint
 - These are usually rough drafts or ways I was thinking of presenting things that I didn't end up going with.
- I just realized that when I export to PDF the hidden slides show up with no distinction from normal slides.
 - I tried to clear these out, but if you find a slide that looks like it doesn't belong or is full of garbage, let me know.

Last week



This week

- Greedy algorithms!
- Builds on our ideas from dynamic programming



Today

- Three examples of greedy algorithms:
 - Activity Selection
 - Job Scheduling
 - Huffman Coding

Example

Activity selection

You can only do one activity at a time, and you want to maximize the number of activities that you do.

What to choose?

CS 161 Class

Math 51 Class

Sleep

CS 161
Section

nar

Progra
team

5 Class

Underwater basket
weaving class

CS110
Class

CS161 study
group

Swimming
lessons

Theory Lunch

Social activity

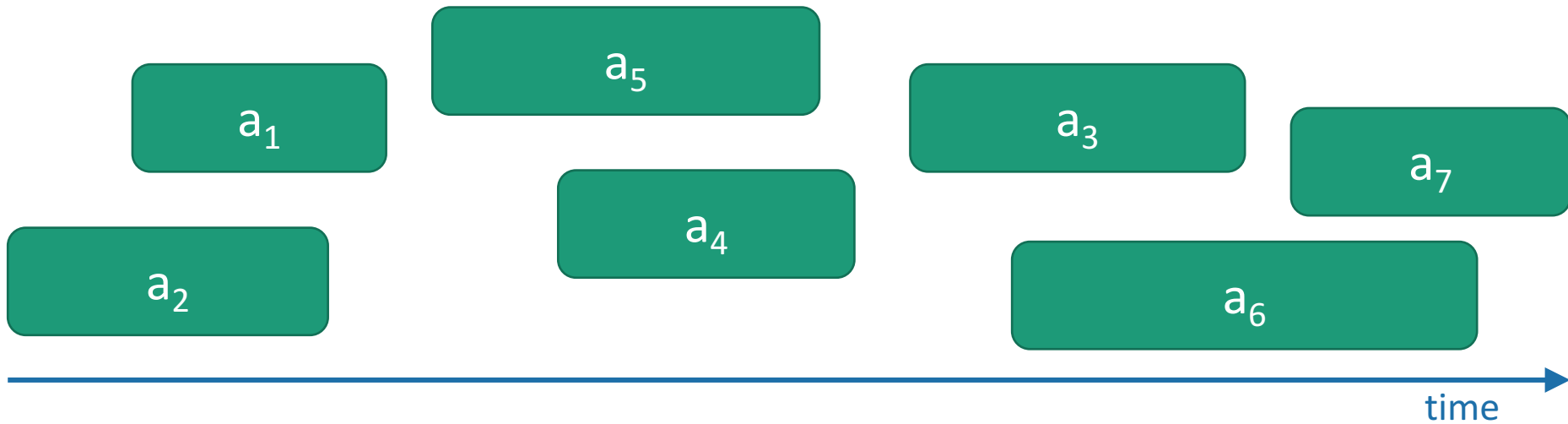
Combinatorics
Seminar

time

Activity selection

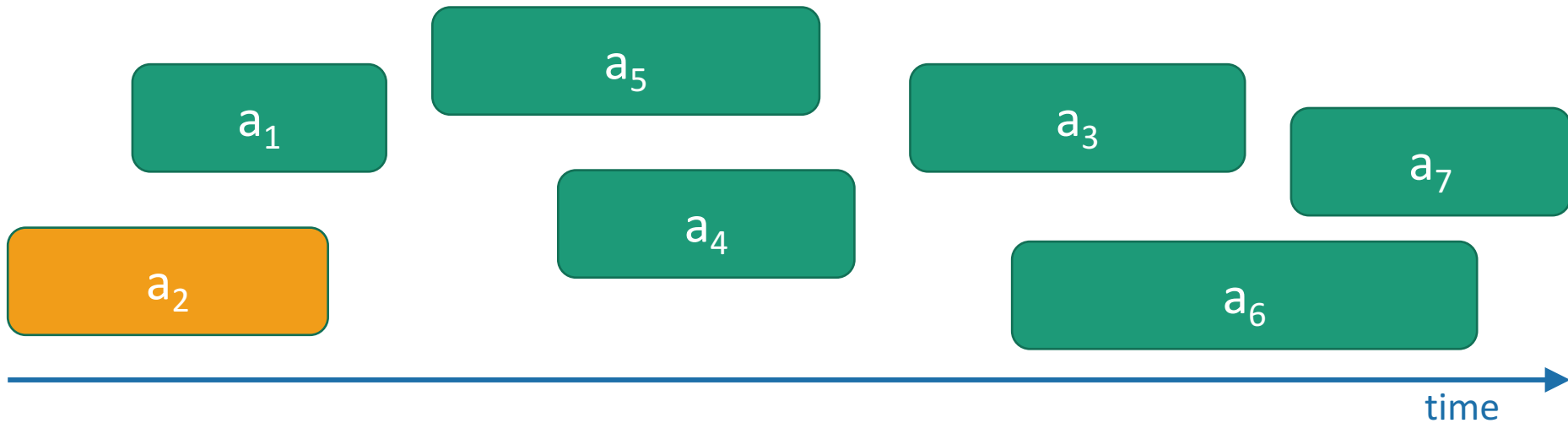
- Input:
 - Activities $a_1, a_2, \dots, a_n$
 - Start times $s_1, s_2, \dots, s_n$
 - Finish times $f_1, f_2, \dots, f_n$
- Output:
 - How many activities can you do today?

Greedy Algorithm



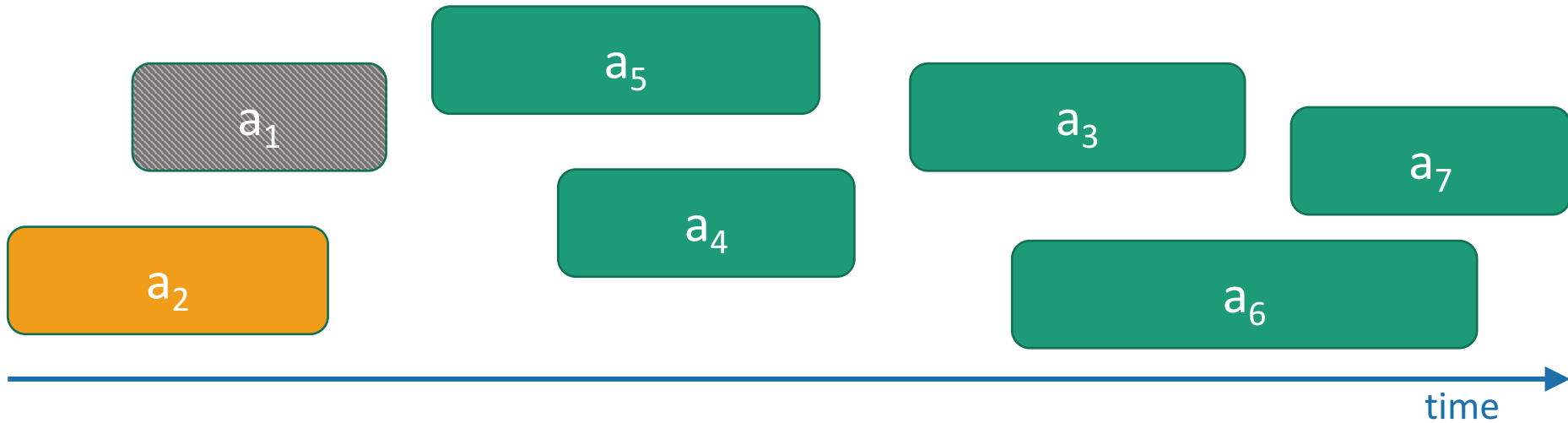
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- Repeat.

Greedy Algorithm



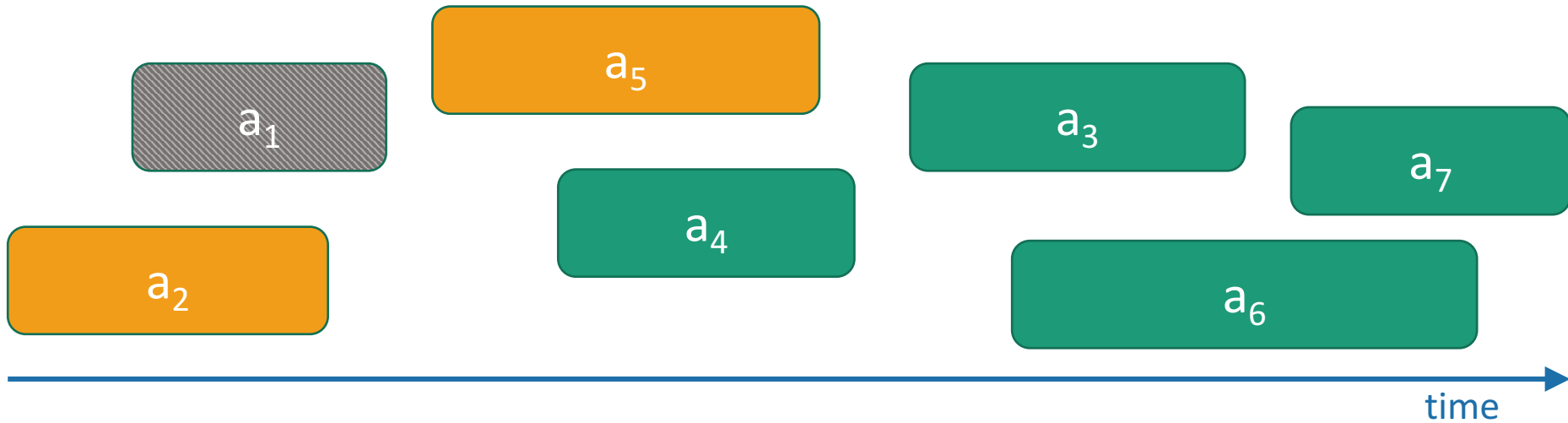
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- Repeat.

Greedy Algorithm



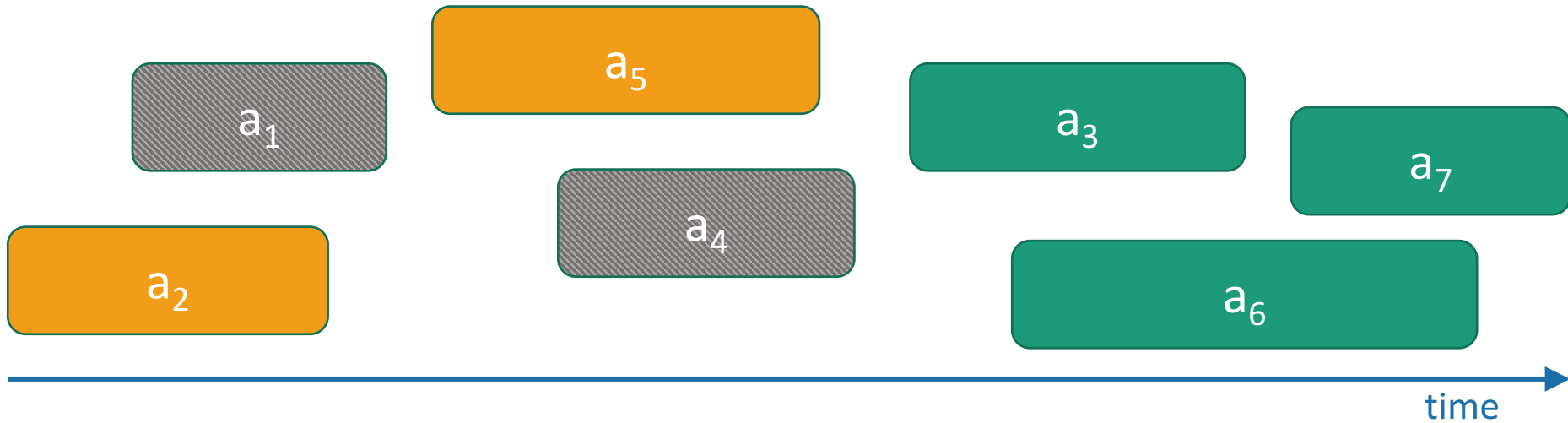
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Greedy Algorithm



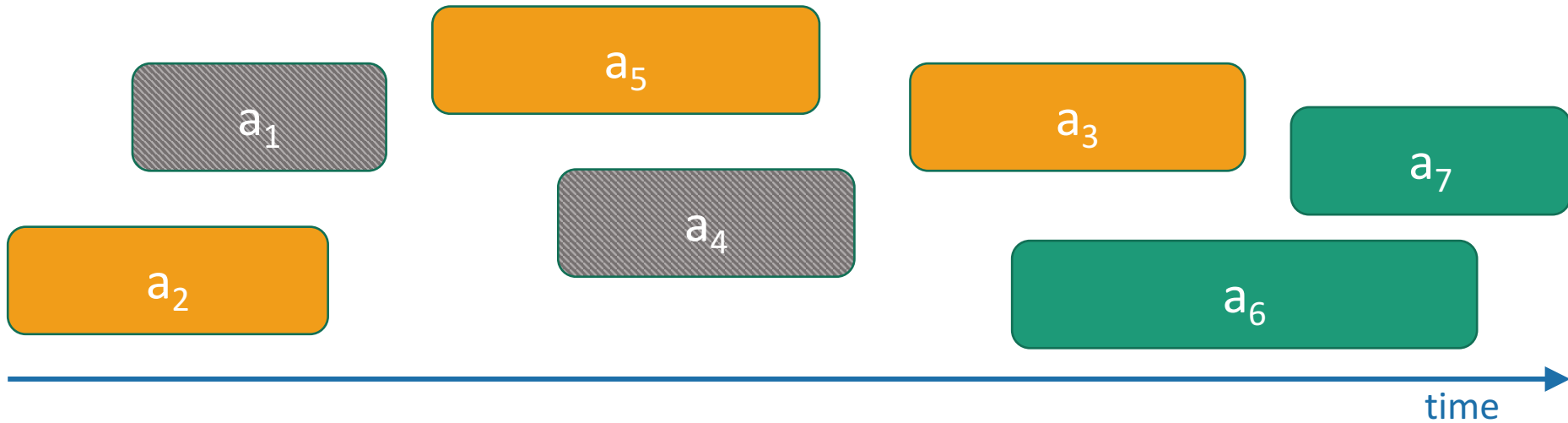
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- Repeat.

Greedy Algorithm



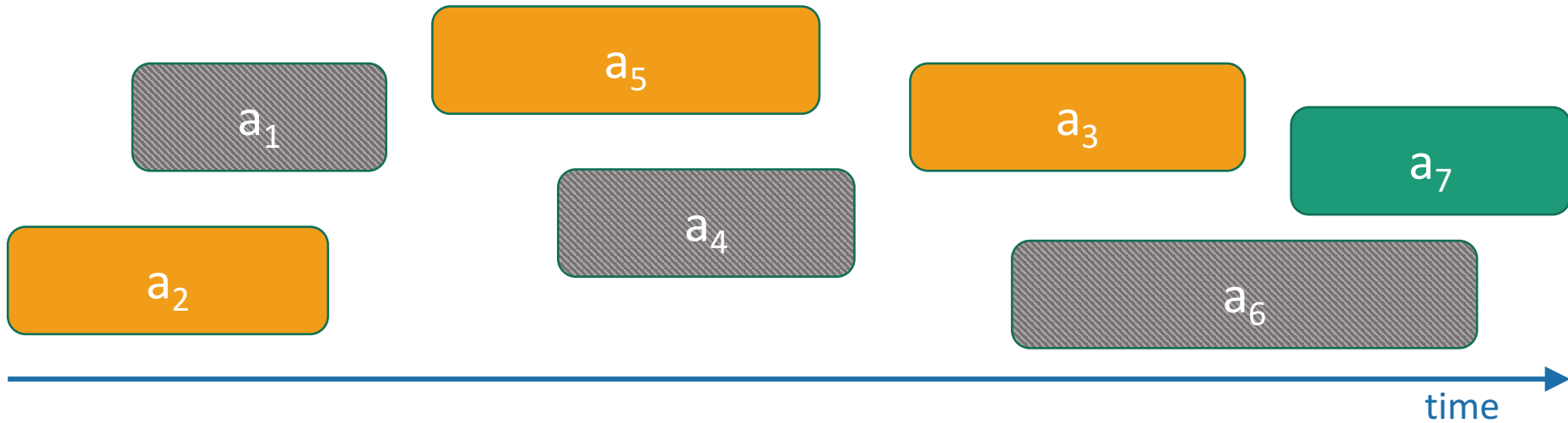
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- Repeat.

Greedy Algorithm



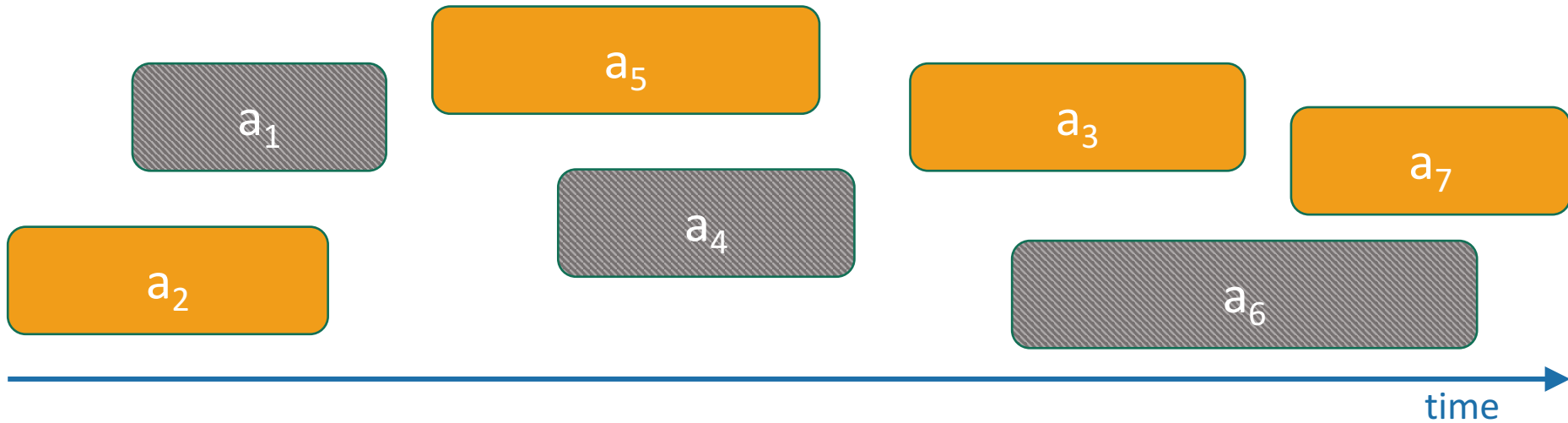
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Greedy Algorithm



- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Greedy Algorithm



- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

That seems like a reasonable thing to do...

- Running time:
 - $O(n)$ if the activities are already sorted by finish time.
 - Otherwise $O(n\log(n))$ if you have to sort them first.
- Does it work?
 - We'll see soon.

This is an example of a **greedy algorithm**

- At each step in the algorithm, make a choice.
 - Hey, I can increase my activity set by one,
 - And leave lots of room for future choices,
 - Let's do that and hope for the best!!!
- **Hope** that at the end of the day, this results in a globally optimal solution.



Three questions

1. Does this greedy algorithm for activity selection work?
2. In general, when are greedy algorithms a good idea?
3. The “greedy” approach is often the first you’d think of...
 - Why are we getting to it now, in Week 8?

Answers

1. Does this greedy algorithm for activity selection work?
 - Yes.
2. In general, when are greedy algorithms a good idea?
 - When they exhibit especially nice optimal substructure.
3. The “greedy” approach is often the first you’d think of...
 - Why are we getting to it now, in Week 8?
 - Like dynamic programming!
 - (Which we did in Week 7).
 - Proving that greedy algorithms work is often not so easy.

DP view of activity selection

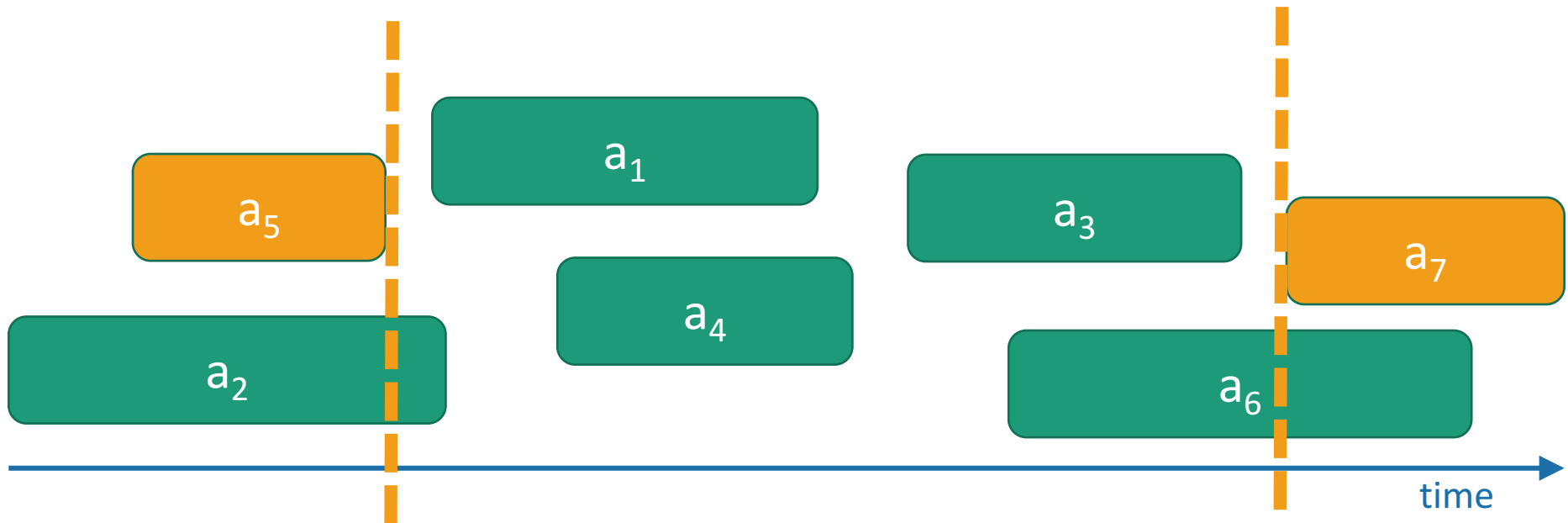
Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure. 
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Optimal substructure

- Subproblems:

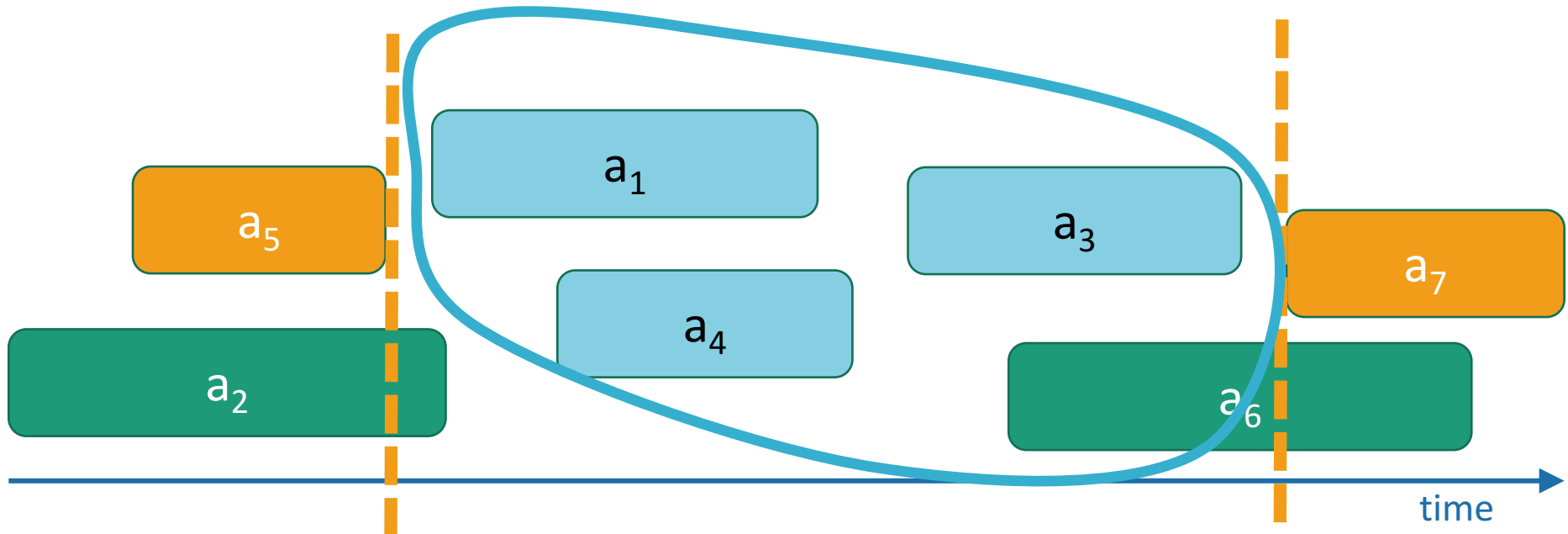
- $A[i,j]$ = number of activities you can squeeze in after Activity i finishes and before Activity j starts



Optimal substructure

- Subproblems:

- $A[i,j]$ = number of activities you can squeeze in after Activity i finishes and before Activity j starts



- $A[5,7]$ = solution to **this subproblem**

Recipe for applying Dynamic Programming

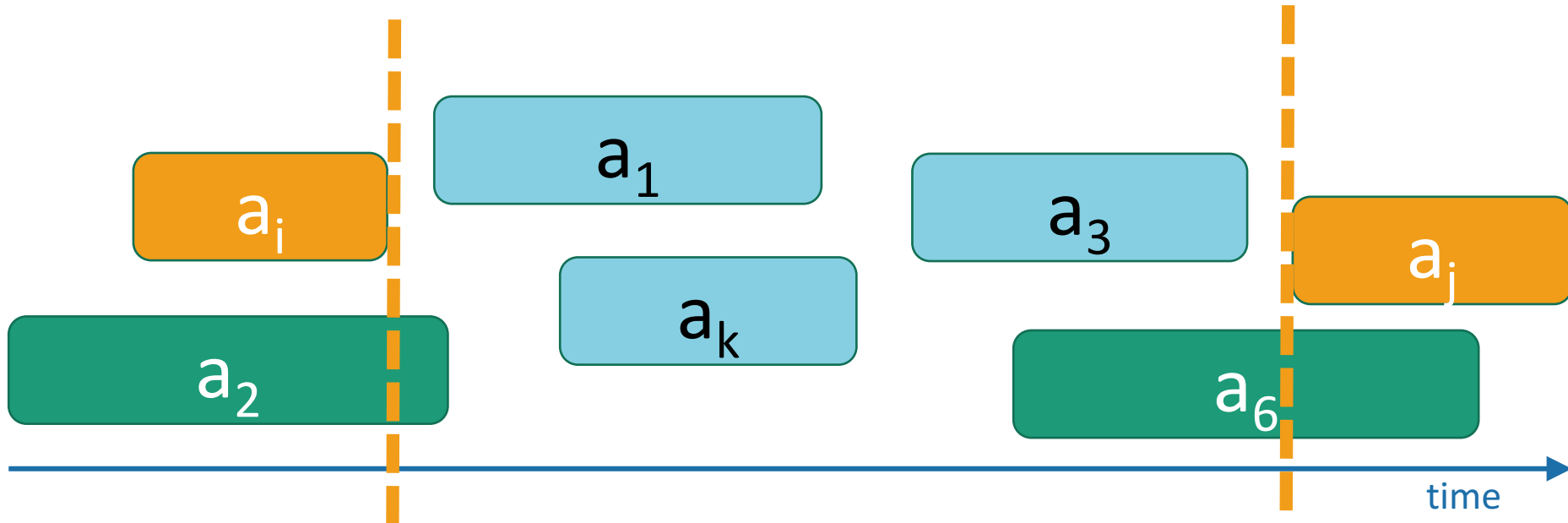
- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.



This satisfies a nice recursive relationship

$A[i,j]$ = number of activities you can squeeze in after Activity i finishes and before Activity j starts

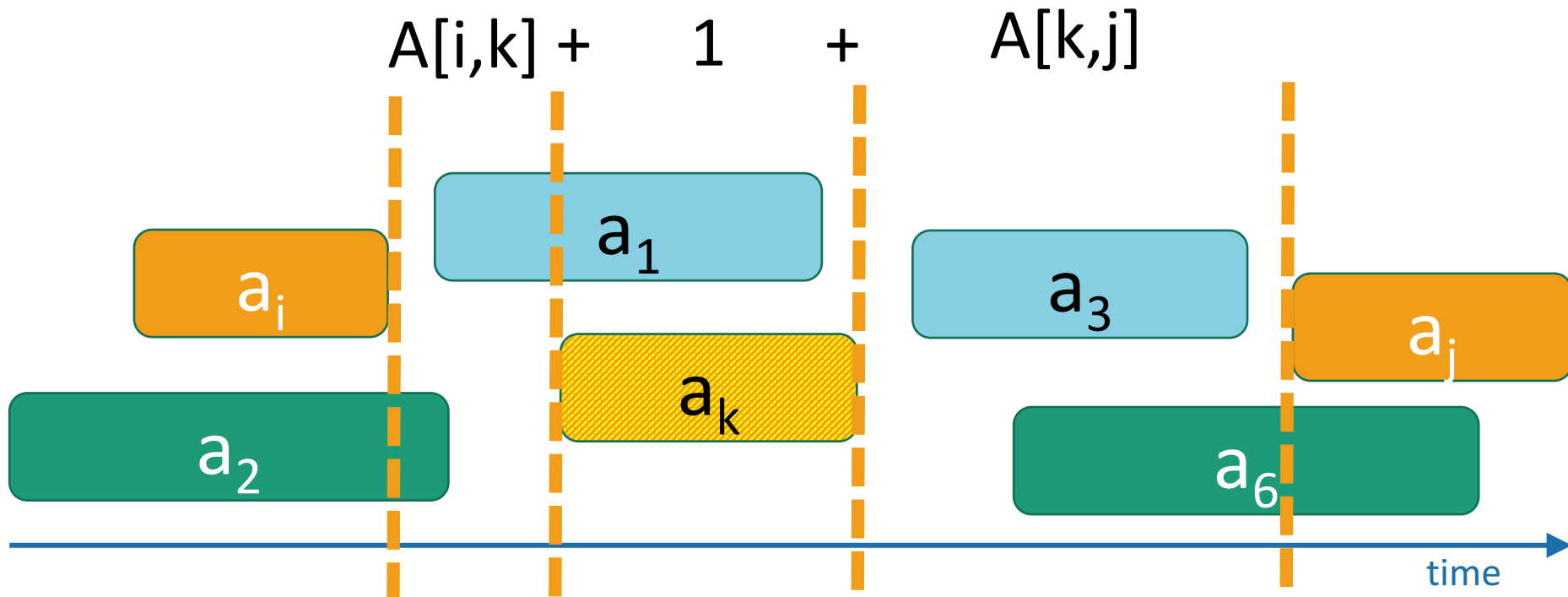
- $A[i,j] = \max_k \{ A[i,k] + 1 + A[k,j] \}$
 - The maximum is over all k so that Activity k fits in between Activities i and j .



This satisfies a nice recursive relationship

$A[i,j]$ = number of activities you can squeeze in after Activity i finishes and before Activity j starts

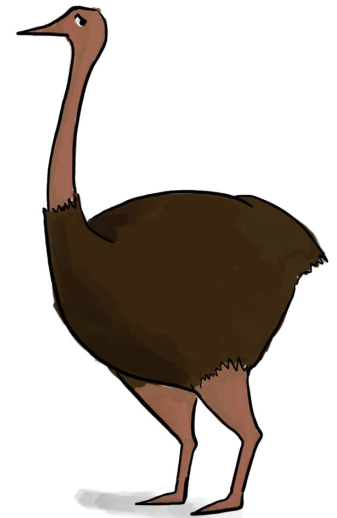
- $A[i,j] = \max_k \{ A[i,k] + 1 + A[k,j] \}$
 - The maximum is over all k so that Activity k fits in between Activities i and j .



We could turn this into a DP algorithm

- .Would take time something like $O(n^3)$
 - Fill out an n -by- n table.
 - For each entry search over maybe n possibilities for k .
- But this would be wasteful!
 - we just saw an algorithm that takes time $O(n \log(n))$, if it's correct...

Try it!
It builds character!



$A[i,j]$ = number of activities you can squeeze in after Activity i finishes and before Activity j starts

The thing that's wasteful

- Actually, we **should know in advance** what subproblem to look at.
- Lemma:
 - Suppose that k is the activity you can squeeze in after i **with the smallest finishing time**.
 - Then there is an **optimal solution to $A[i..n+1]$** that **extends the optimal solution to $A[k..n+1]$** .

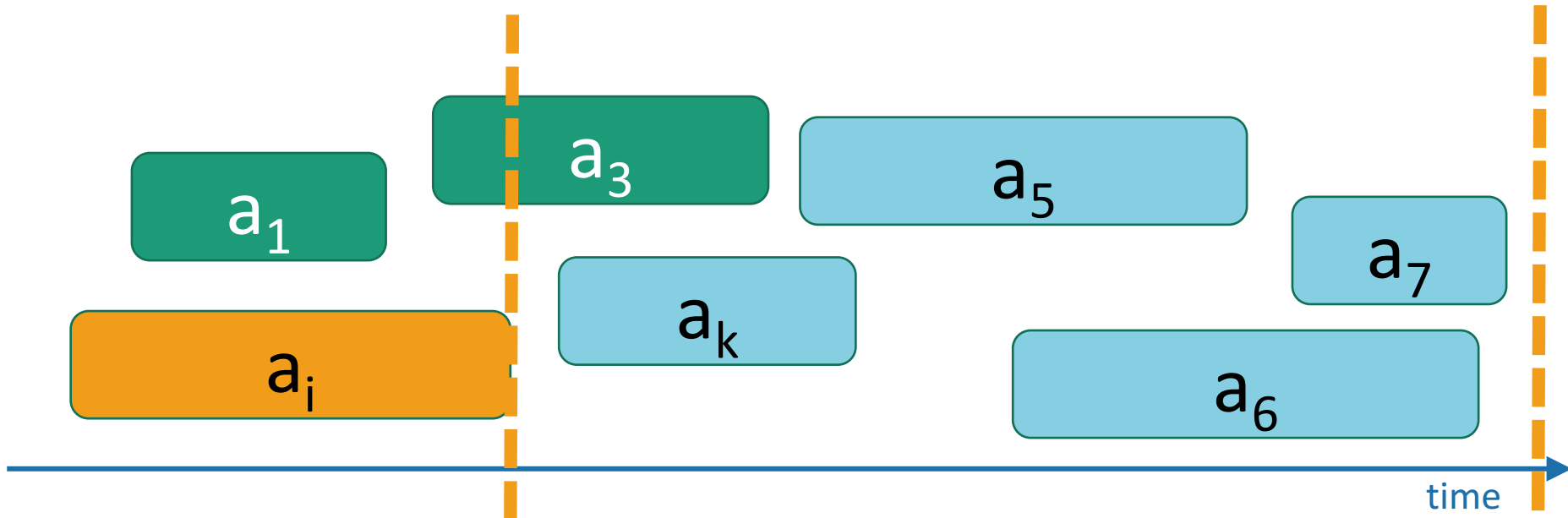
This is abusing notation...technically $A[i..n+1]$ is a value, not a problem.

Let's add an additional activity a_{n+1} that starts "tomorrow".

$A[i,j]$ = number of activities you can squeeze in after Activity i finishes and before Activity j starts

Lemma

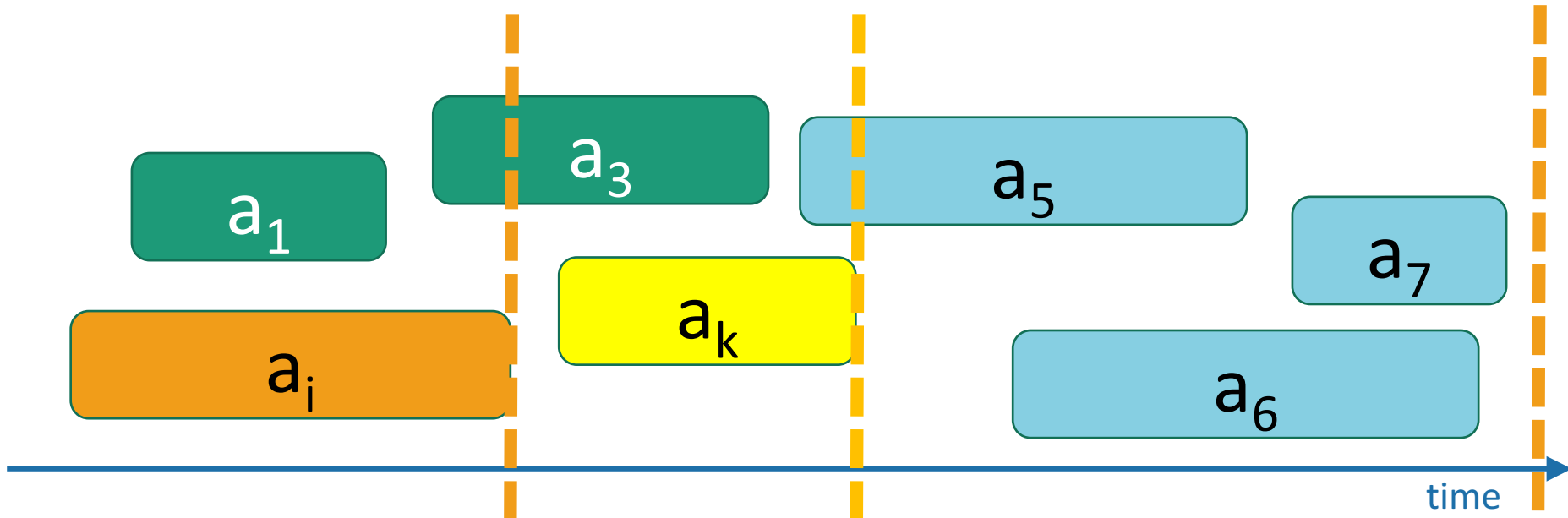
- Suppose that k is the activity you can squeeze in after i **with the smallest finishing time**.
- Then there is an **optimal solution to $A[i..n+1]$** that **extends the optimal solution to $A[k..n+1]$** .



$A[i,j]$ = number of activities you can squeeze in after Activity i finishes and before Activity j starts

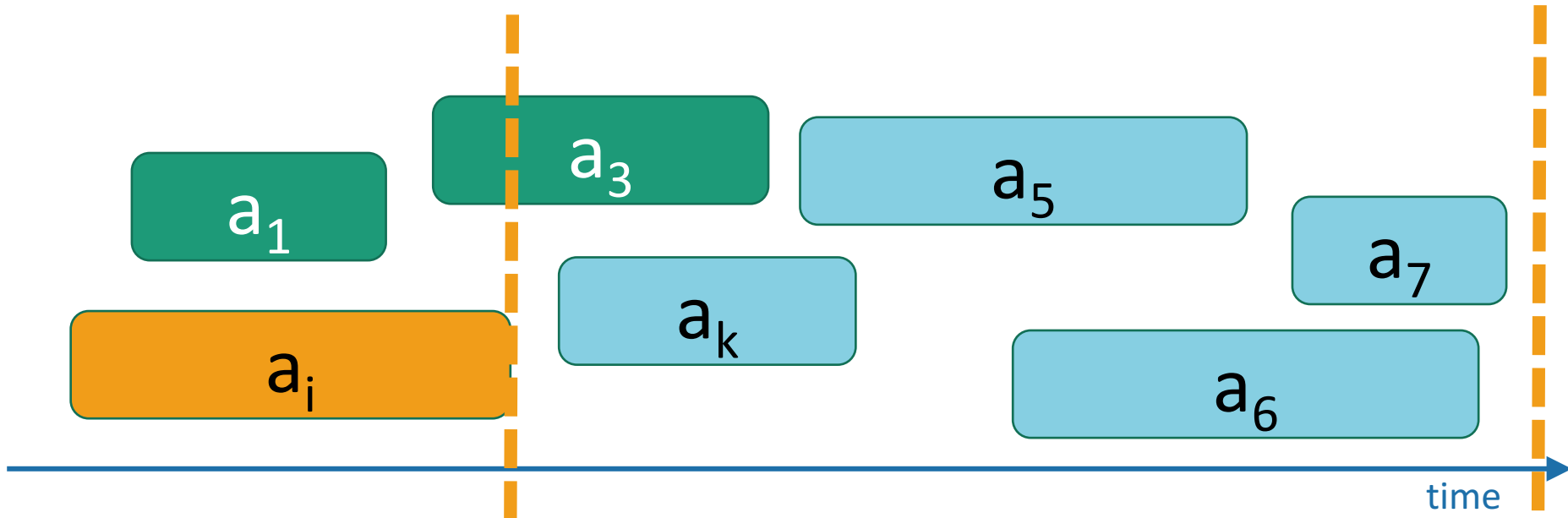
Lemma

- Suppose that k is the activity you can squeeze in after i **with the smallest finishing time**.
- Then there is an **optimal solution to $A[i..n+1]$** that **extends the optimal solution to $A[k..n+1]$** .



Proof

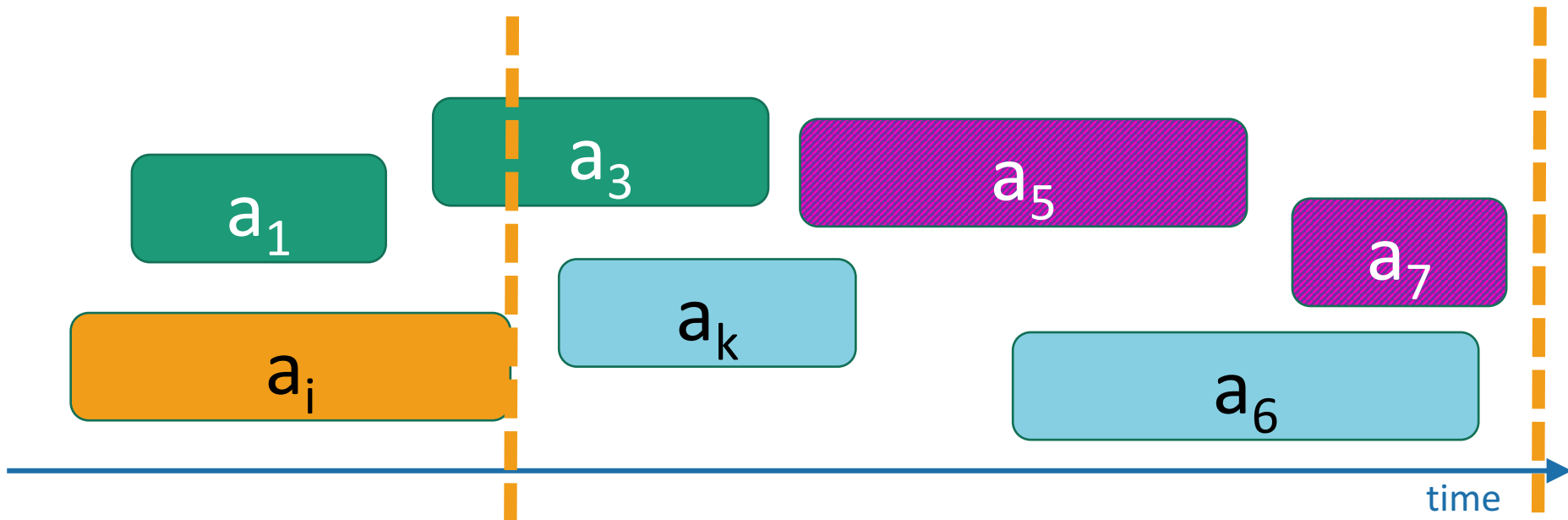
- Suppose that k is the activity you can squeeze in after i **with the smallest finishing time**.
- Then there is an **optimal solution to $A[i..n+1]$** that **extends the optimal solution to $A[k..n+1]$** .



Proof

- Suppose that k is the activity you can squeeze in after i **with the smallest finishing time**.
- Then there is an **optimal solution to $A[i..n+1]$** that **extends the optimal solution to $A[k..n+1]$** .

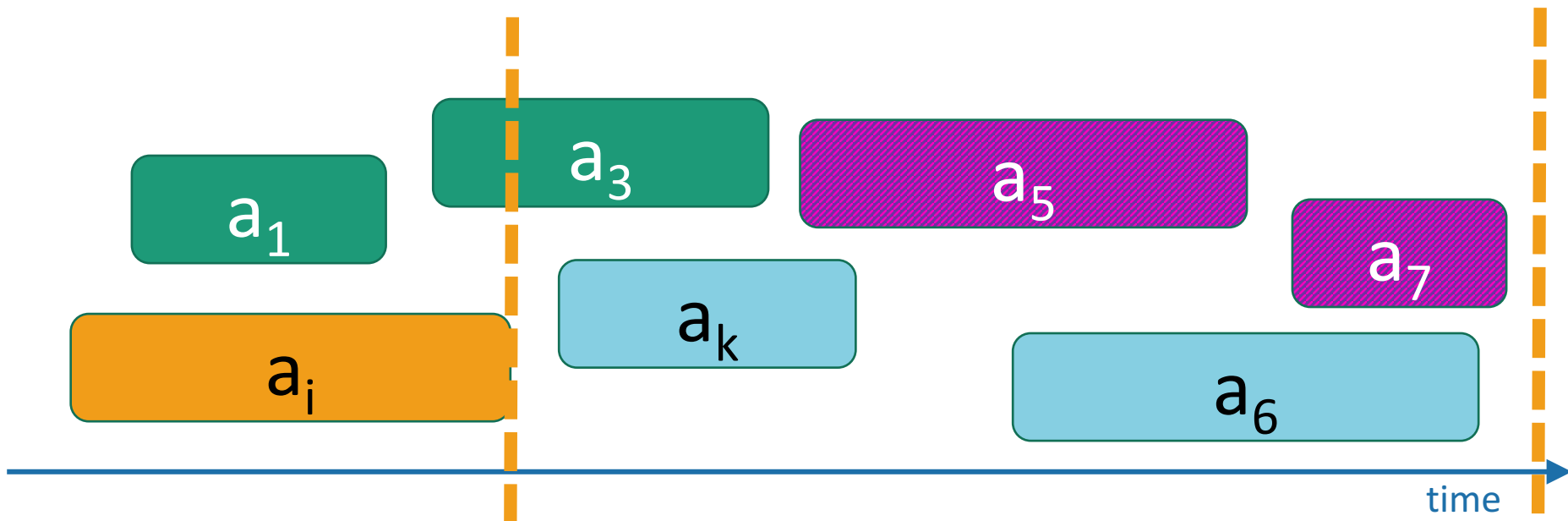
- Suppose that **this** is an optimal solution to **$A[i..n+1]$**
 - Doesn't involve a_k



Proof

- Suppose that k is the activity you can squeeze in after i **with the smallest finishing time**.
- Then there is an **optimal solution to $A[i..n+1]$** that **extends the optimal solution to $A[k..n+1]$** .

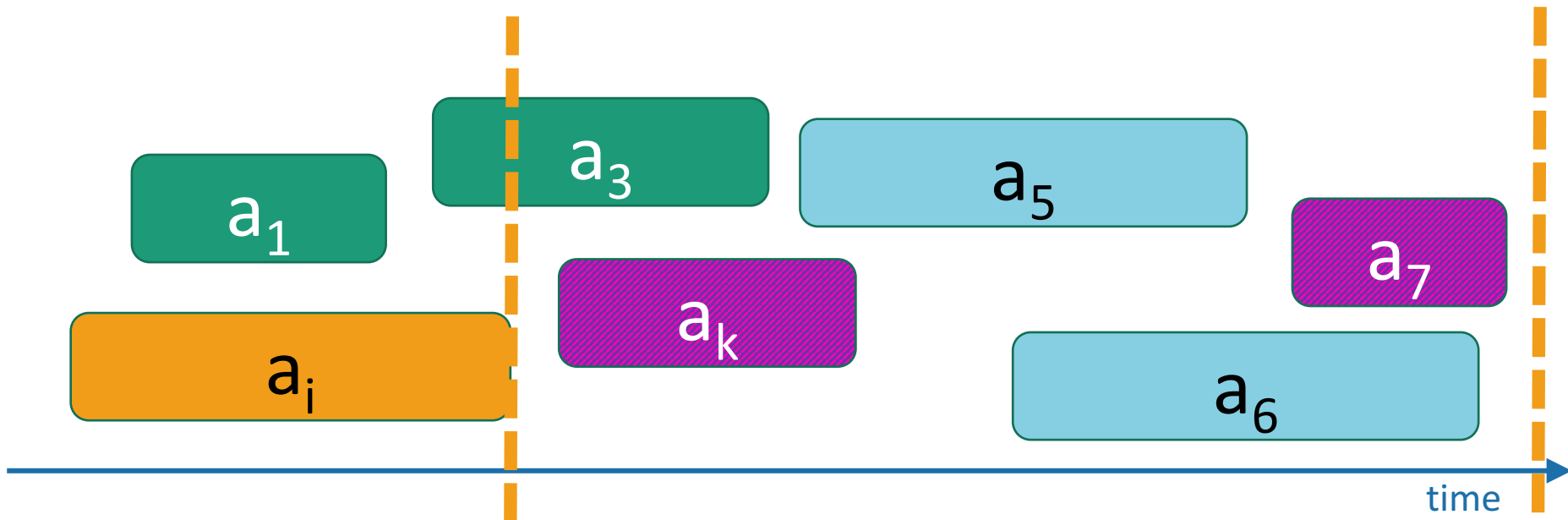
- Suppose that **this** is an optimal solution to **$A[i..n+1]$**
 - Doesn't involve a_k
- **Swap a_k in** for whatever had the smallest finishing time in **that solution**.



Proof

- Suppose that k is the activity you can squeeze in after i **with the smallest finishing time**.
- Then there is an **optimal solution to $A[i..n+1]$** that **extends the optimal solution to $A[k..n+1]$** .

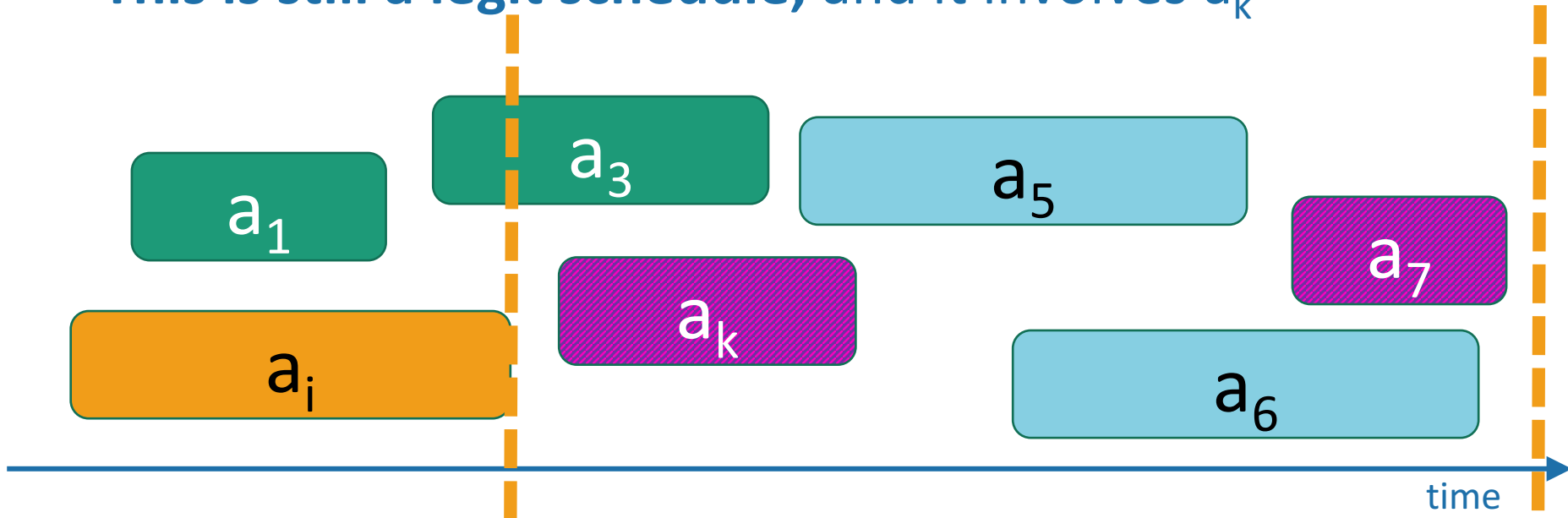
- Suppose that **this** is an optimal solution to **$A[i..n+1]$**
 - Doesn't involve a_k
- **Swap a_k in** for whatever had the smallest finishing time in **that solution**.



Proof

- Suppose that k is the activity you can squeeze in after i **with the smallest finishing time**.
- Then there is an **optimal solution to $A[i..n+1]$** that **extends the optimal solution to $A[k..n+1]$** .

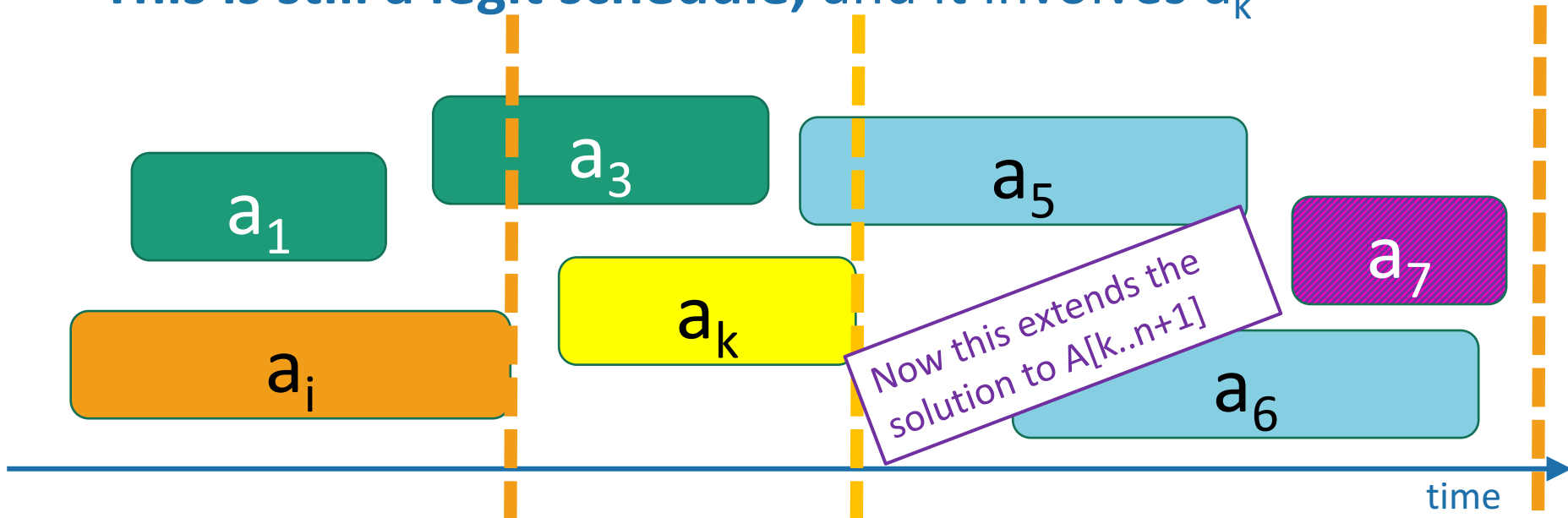
- Suppose that **this** is an optimal solution to **$A[i..n+1]$**
 - Doesn't involve a_k
- **Swap a_k in** for whatever had the smallest finishing time in **that solution**.
- **This is still a legit schedule**, and it involves a_k



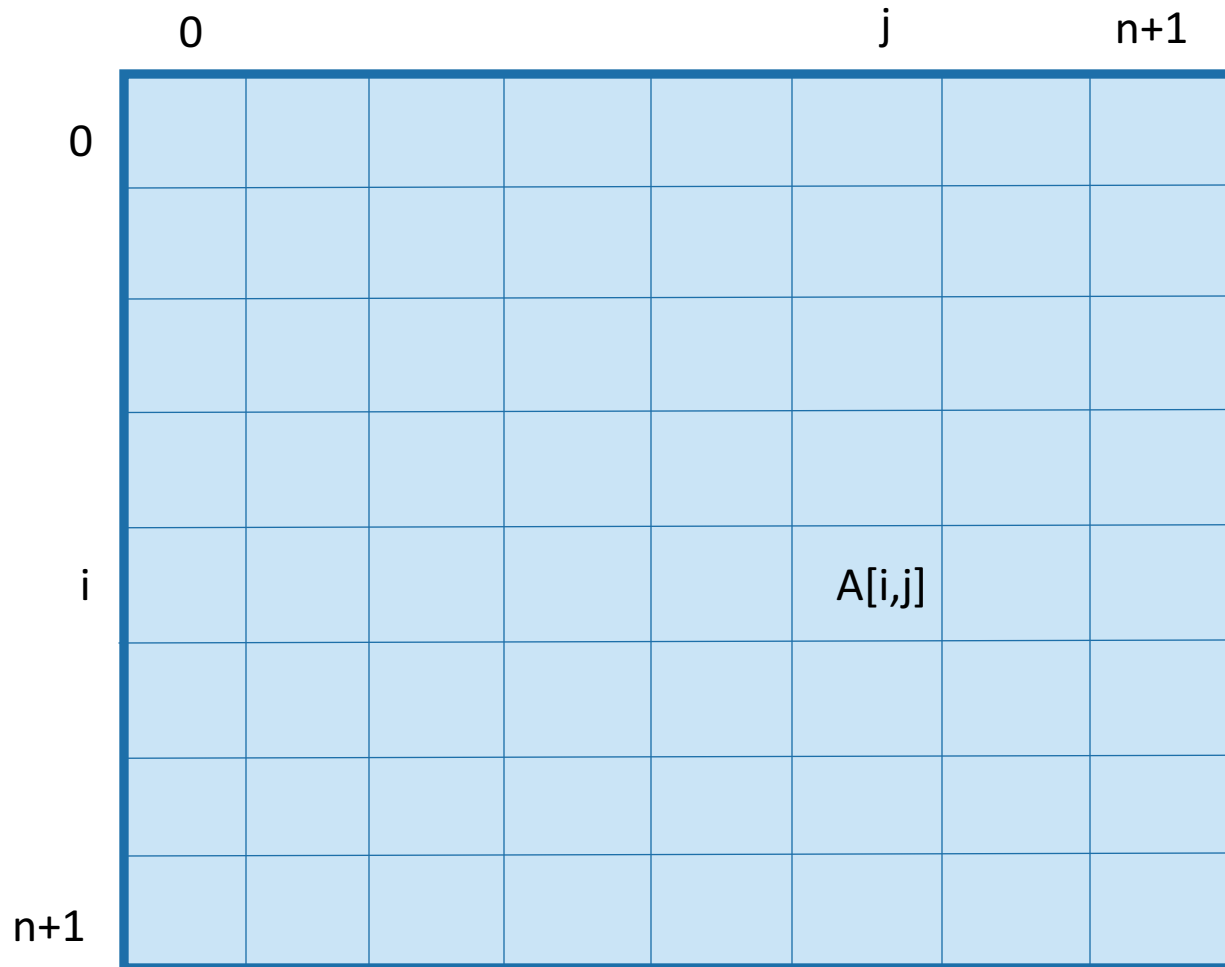
Proof

- Suppose that k is the activity you can squeeze in after i **with the smallest finishing time**.
- Then there is an **optimal solution to $A[i..n+1]$** that **extends the optimal solution to $A[k..n+1]$** .

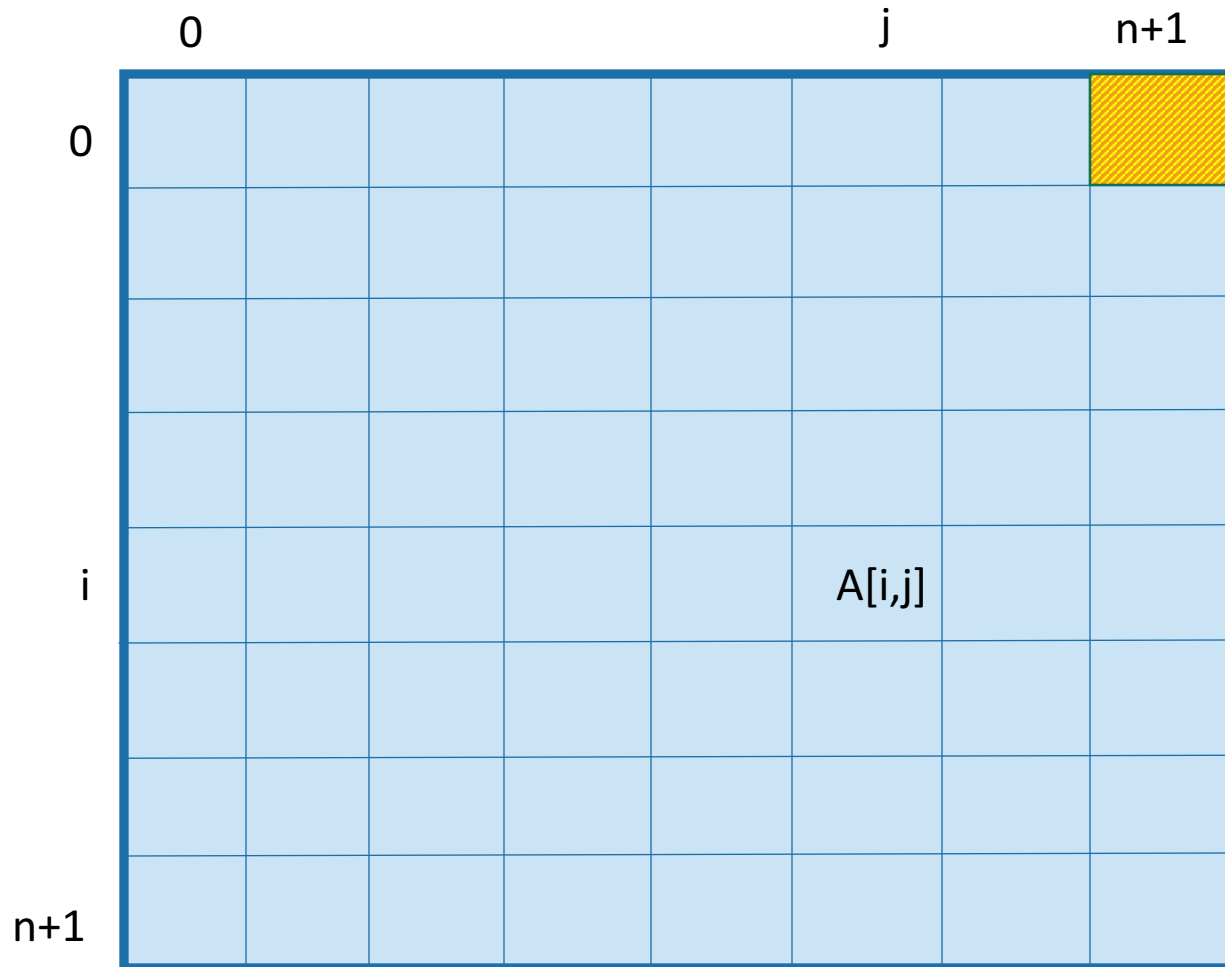
- Suppose that **this** is an optimal solution to **$A[i..n+1]$**
 - Doesn't involve a_k
- **Swap a_k in** for whatever had the smallest finishing time in **that solution**.
- **This is still a legit schedule**, and it involves a_k



This means that DP would have been wasteful.

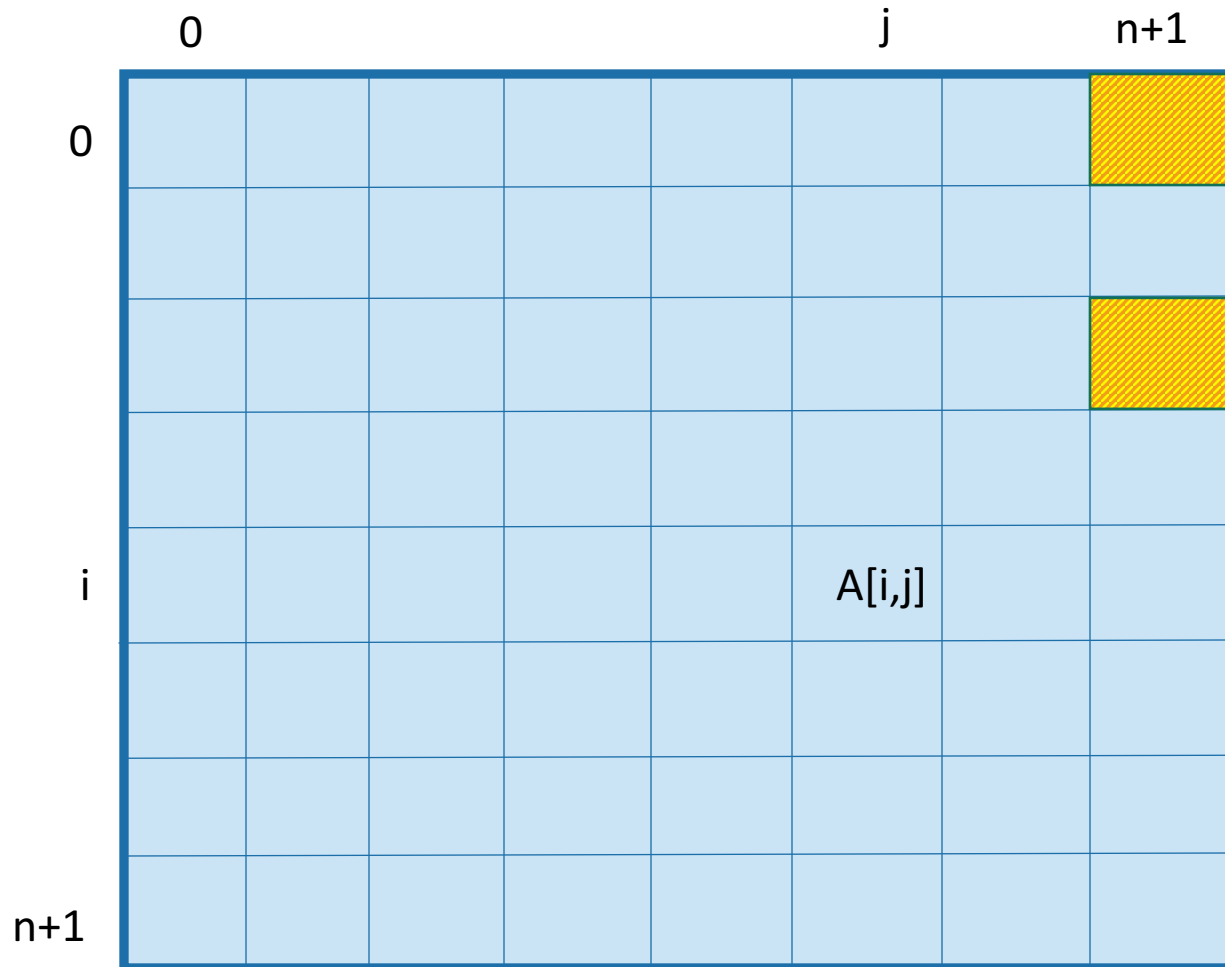


This means that DP would have been wasteful.



$A[0, n+1]$ is the return value we wanted.

This means that DP would have been wasteful.



$A[0, n+1]$ is the return value we wanted.

We should know ahead of time that it only depends on $A[2, n+1]$

This means that DP would have been wasteful.

	0					j		n+1
0								
i						A[i,j]		
n+1								

$A[0,n+1]$ is the return value we wanted.

We should know ahead of time that it only depends on $A[2,n+1]$

etc.

This means that DP would have been wasteful.

	0					j		n+1
0								
i						A[i,j]		
n+1								

$A[0,n+1]$ is the return value we wanted.

We should know ahead of time that it only depends on $A[2,n+1]$

etc.

This means that DP would have been wasteful.

	0					j		n+1
0								
i						A[i,j]		
n+1								

$A[0,n+1]$ is the return value we wanted.

We should know ahead of time that it only depends on $A[2,n+1]$

etc.

This means that DP would have been wasteful.

	0					j		n+1
0								
i						A[i,j]		
n+1								

$A[0,n+1]$ is the return value we wanted.

We should know ahead of time that it only depends on $A[2,n+1]$

etc.

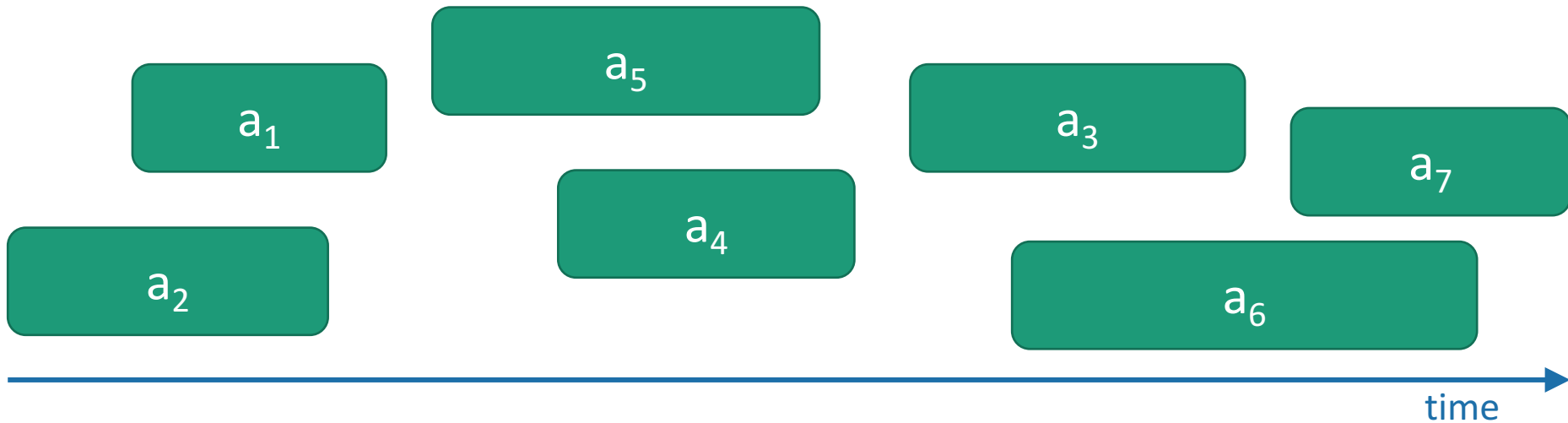
There's no reason we have to look at the whole table!

Instead, let's use this insight to make a **greedy algorithm**

- Suppose the activities are sorted by finishing time
 - if not, sort them.
- `mySchedule = []`
- **for** $k = 1, \dots, n$:
 - if I can fit in **Activity** k after the last thing in `mySchedule`:
 - `mySchedule.append(Activity k)`
- return `mySchedule`

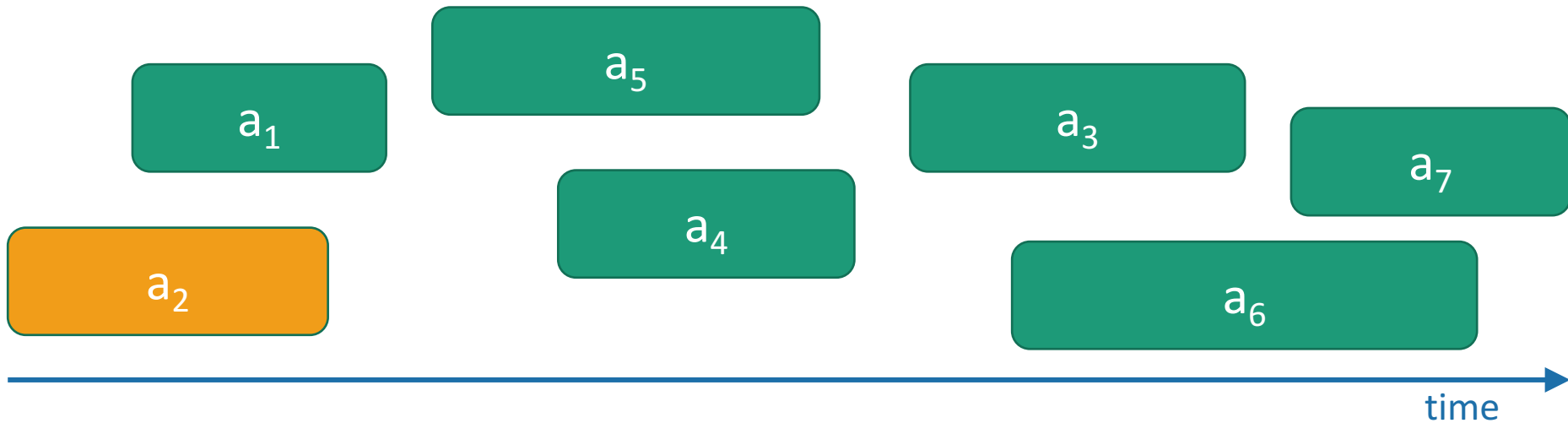
This is the same thing
we saw before

Greedy Algorithm



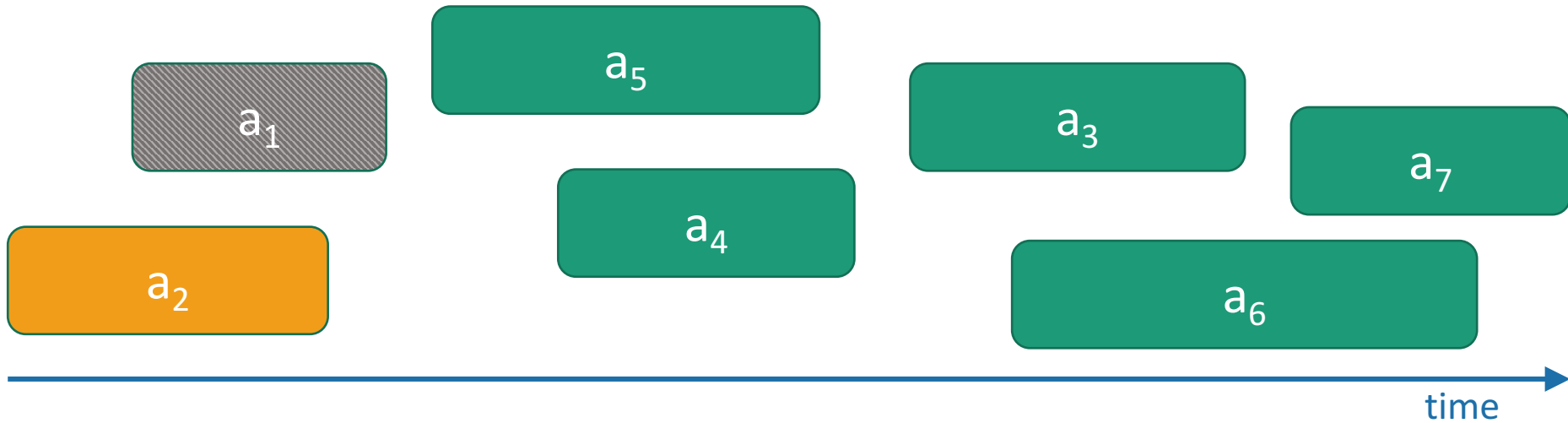
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Greedy Algorithm



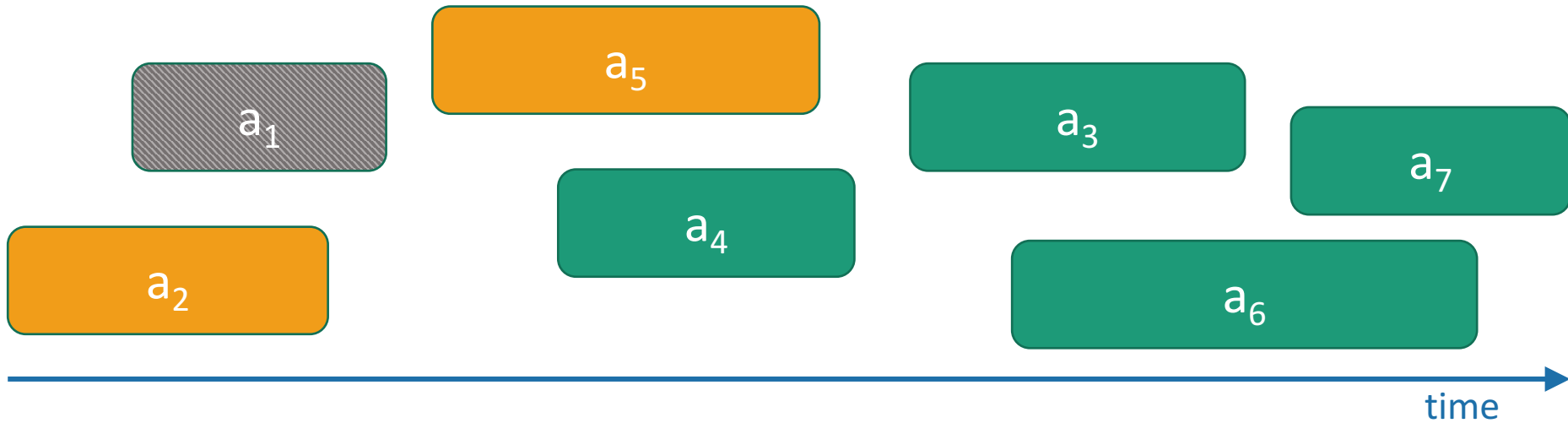
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- Repeat.

Greedy Algorithm



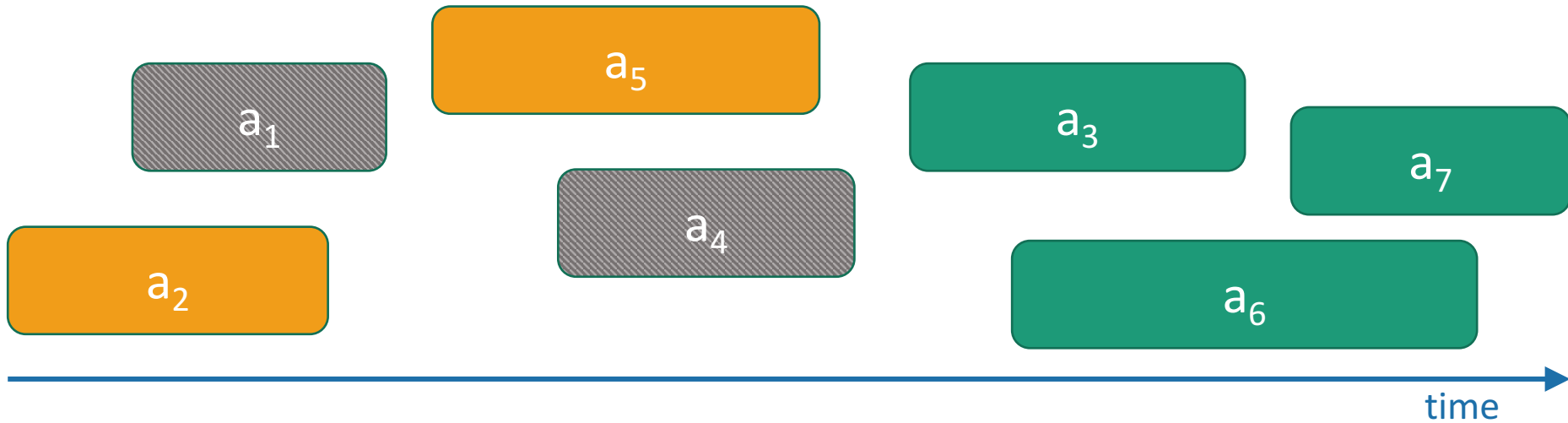
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Greedy Algorithm



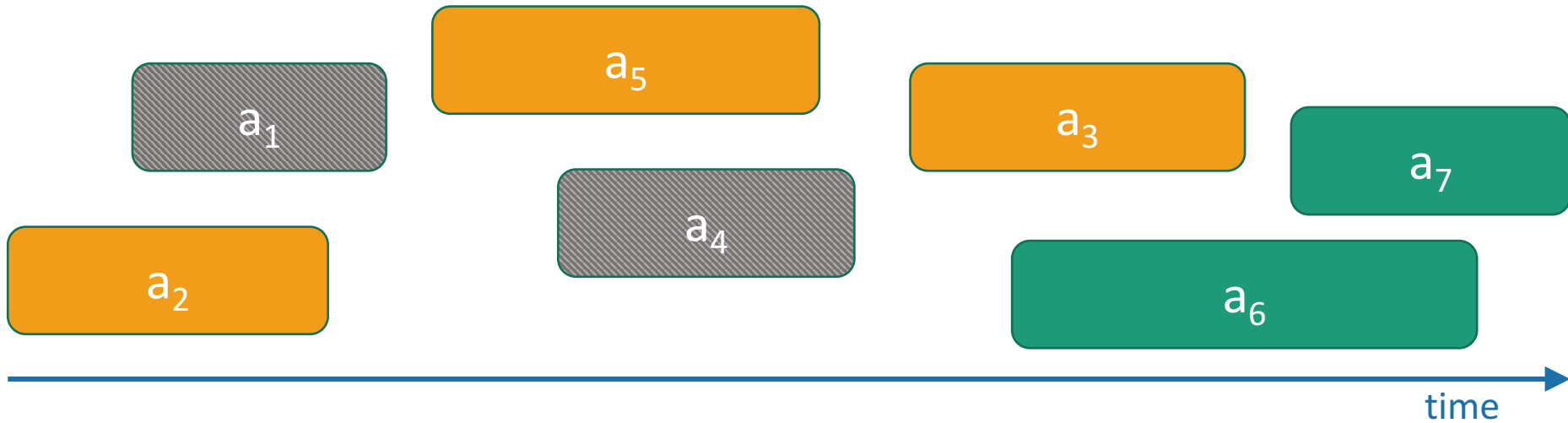
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Greedy Algorithm



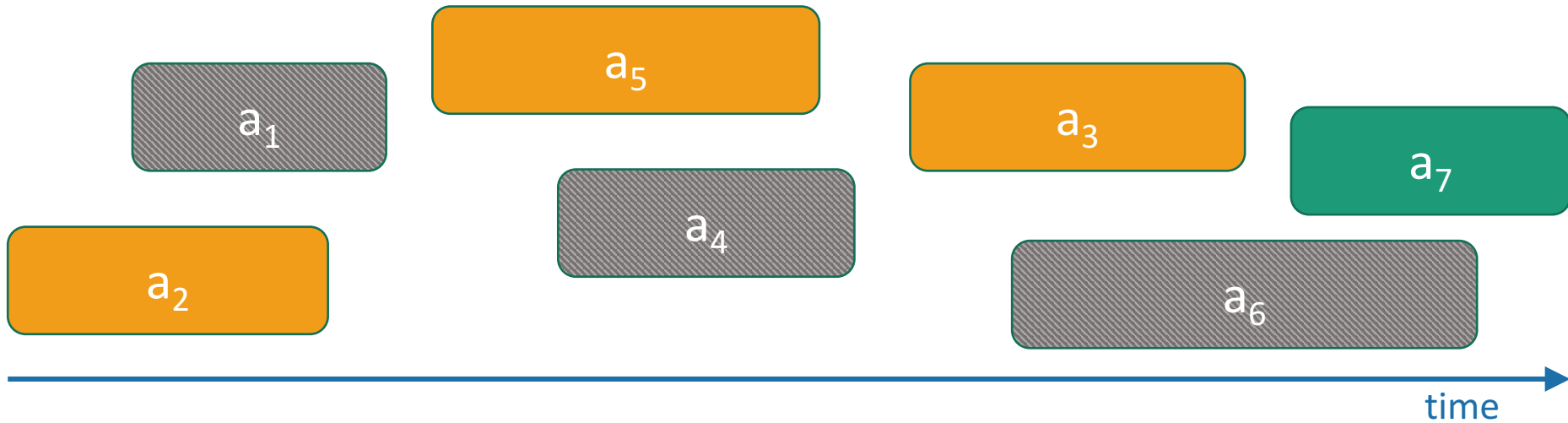
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Greedy Algorithm



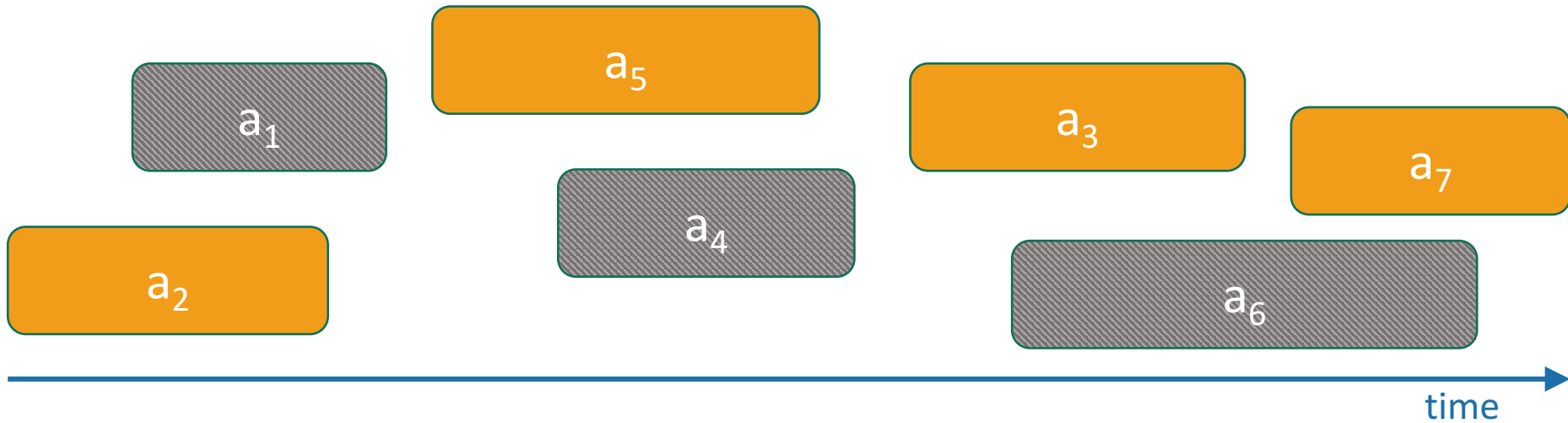
- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Greedy Algorithm



- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Greedy Algorithm



- Pick the activity you can add
 - that has the **smallest finish time**.
- Include it in your activity list.
- **Repeat.**

Why does this work?

- At each step, **we make a choice**
 - Include activity k
- We can show that this choice **will never rule out an optimal solution.**
 - Formally: There is an optimal solution to $A[i..n+1]$ that contains $A[k..n+1]$.
- So when we reach the **end of the argument:**
 - we haven't ruled out an optimal solution
 - and we only have one solution left
 - **so it must be optimal.**

Answers

1. Does this greedy algorithm for activity selection work?

- Yes.

2. In general, when are greedy algorithms a good idea?

- When they exhibit especially nice optimal substructure.

In particular, when each big problem
depends on only one sub-problem.

3. The “greedy” approach is often the first you’d think of...

- Why are we getting to it now, in Week 8?

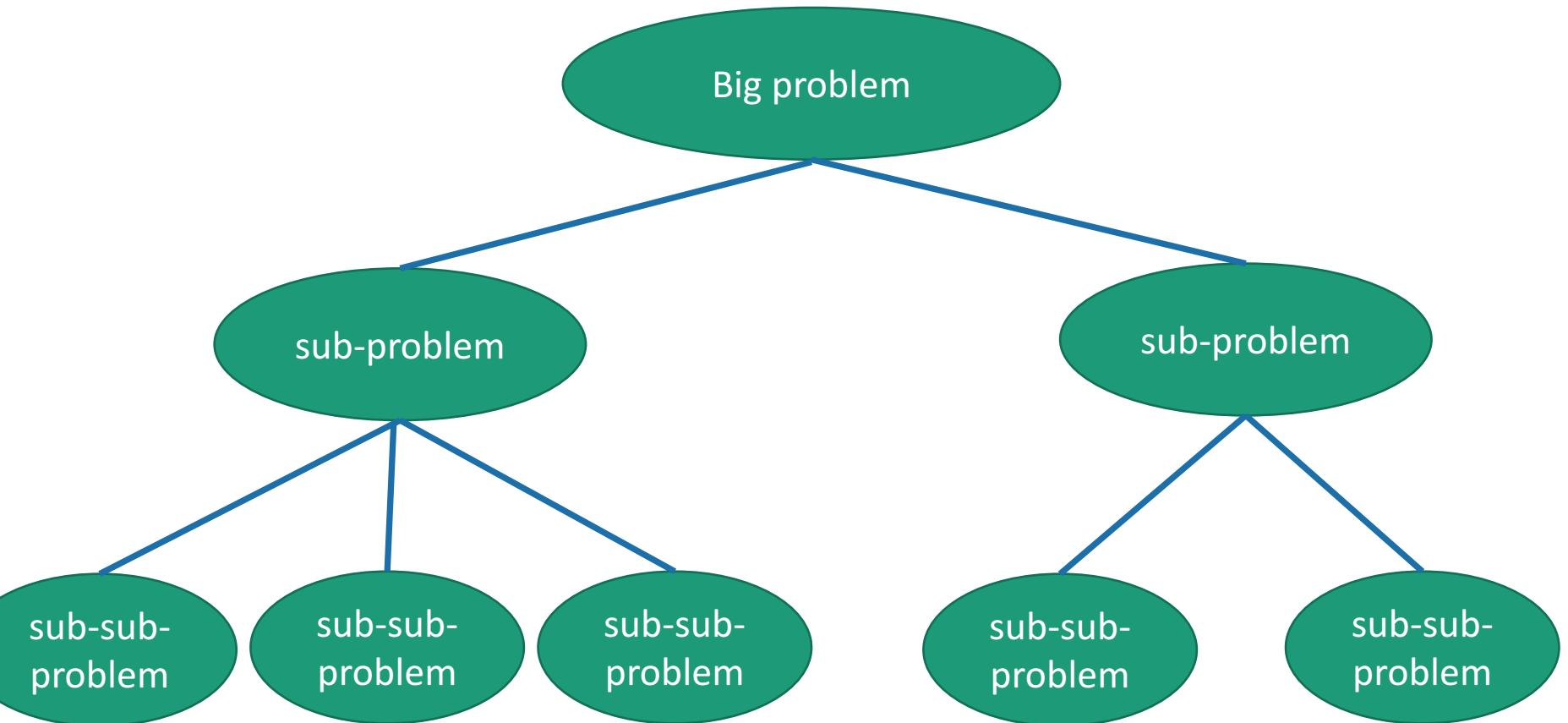
- Like dynamic programming!

- (Which we did in Week 7).

- Proving that greedy algorithms work is often not so easy.

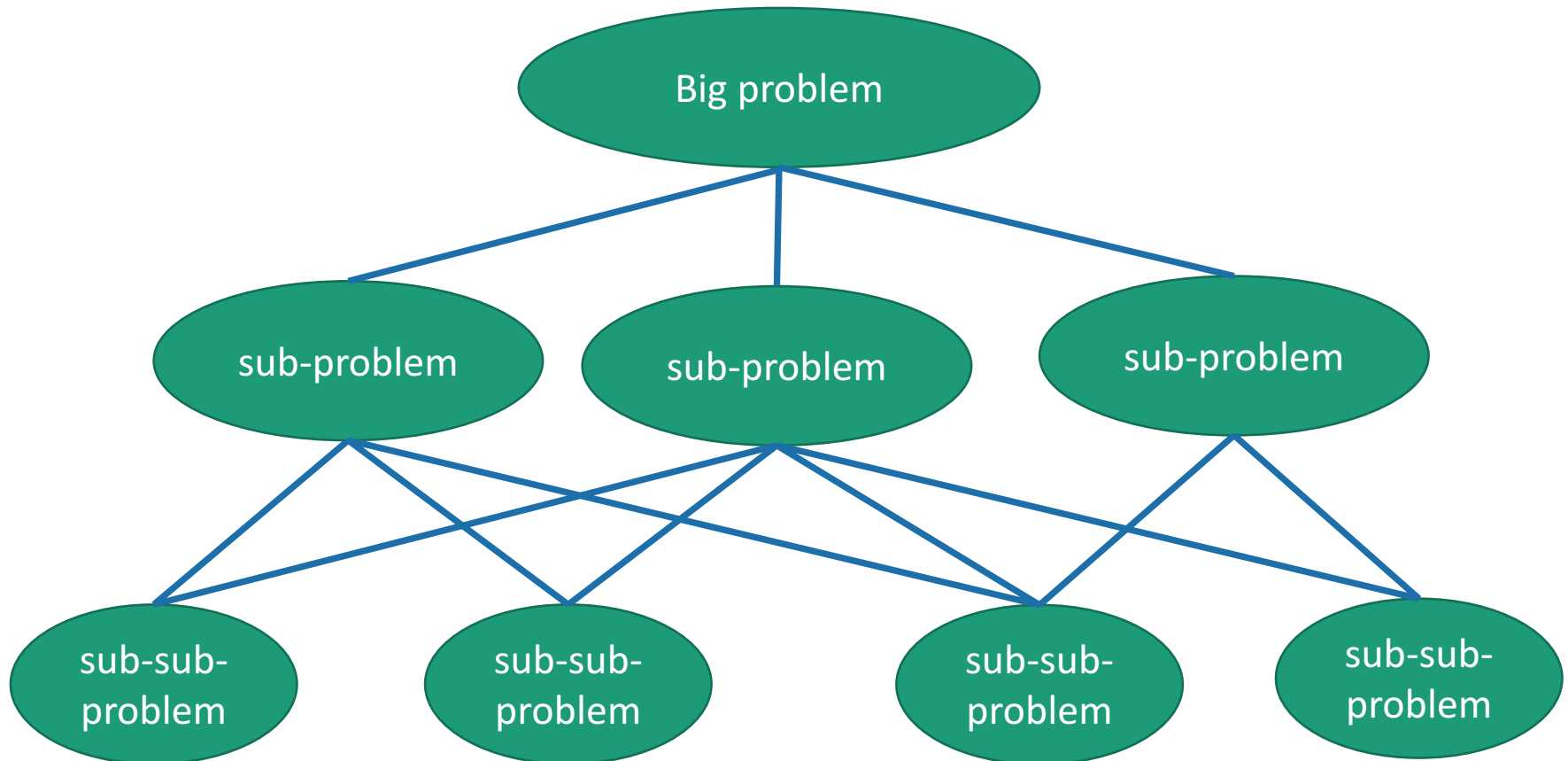
Sub-problem graph view

- Divide-and-conquer:



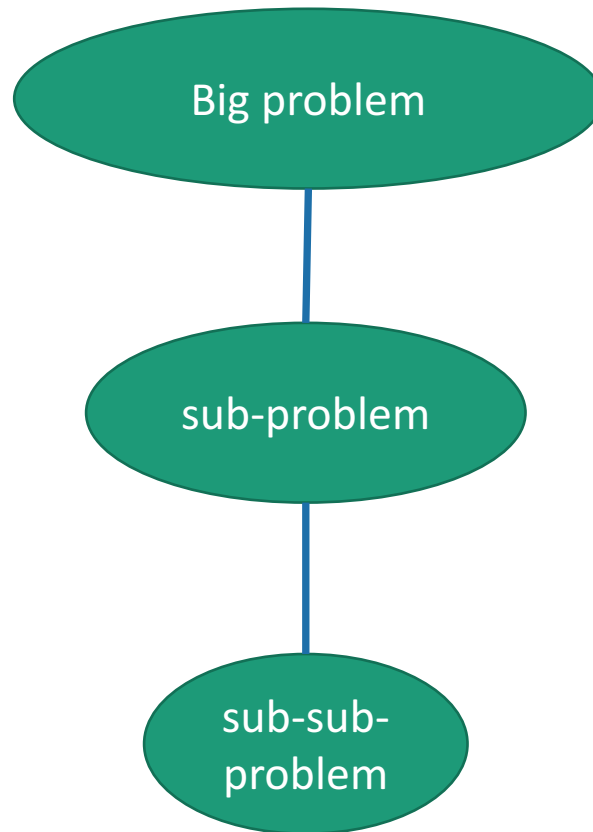
Sub-problem graph view

- Dynamic Programming:



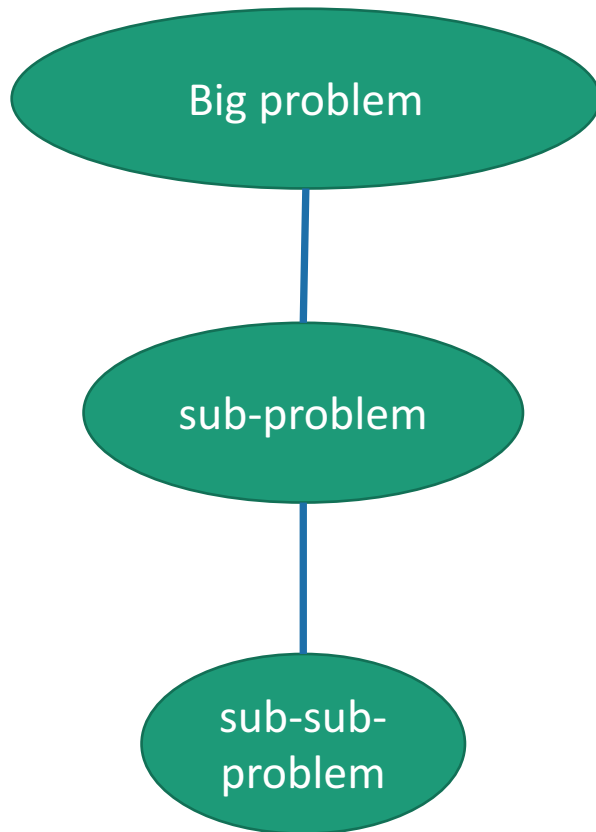
Sub-problem graph view

- Greedy algorithms:



Sub-problem graph view

- Greedy algorithms:



- Not only is there **optimal sub-structure**:
 - optimal solutions to a problem are made up from optimal solutions of sub-problems
- but each problem **depends on only one sub-problem**.

What have we learned?

- If we come up with a DP solution, and it turns out that we really only care about one sub-problem, then maybe we can use a greedy algorithm.
- One example was activity selection.
- In order to come up with a greedy algorithm, we:
 - Made a series of choices
 - Proved that our choices will never rule out an optimal solution.
 - Conclude that our solution at the end is optimal.

Let's see a few more examples

Another example: Scheduling

CS161 HW!

Call your parents!

Math HW!

Administrative stuff for your student club!

Econ HW!

Do laundry!

Meditate!

Practice musical instrument!

Read CLRS!

Have a social life!

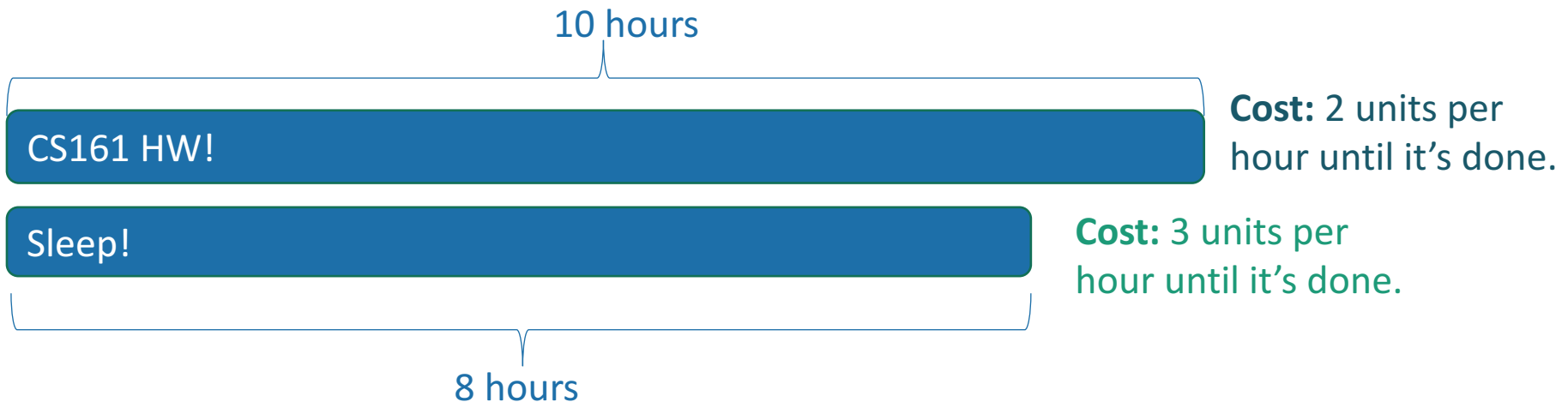
Sleep!

Overcommitted
Stanford Student



Scheduling

- n tasks
- Task i takes t_i hours
- ***Everything is already late!***
 - For every hour that passes until task i is done, pay c_i



- CS161 HW, then Sleep: costs $10 \cdot 2 + (10 + 8) \cdot 3 = 74$ units
- Sleep, then CS161 HW: costs $8 \cdot 3 + (10 + 8) \cdot 2 = 60$ units

Optimal substructure

- This problem breaks up nicely into sub-problems:

Suppose this is the optimal schedule:



Then this must be the optimal schedule on just jobs A and B.

How to use this optimal sub-structure to design a greedy algorithm?

- We make a **series of choices**.
- We show that, at each step, our choice **won't rule out an optimal solution** at the end of the day.
- After we've made all our choices, we haven't ruled out an optimal solution, **so we must have found one**.

Of all these jobs, which one(s) is it safe to choose first?

Which won't rule out an optimal solution?

Job A

Job B

Job C

Job D

Head-to-head

A then B is better than **B then A** when:

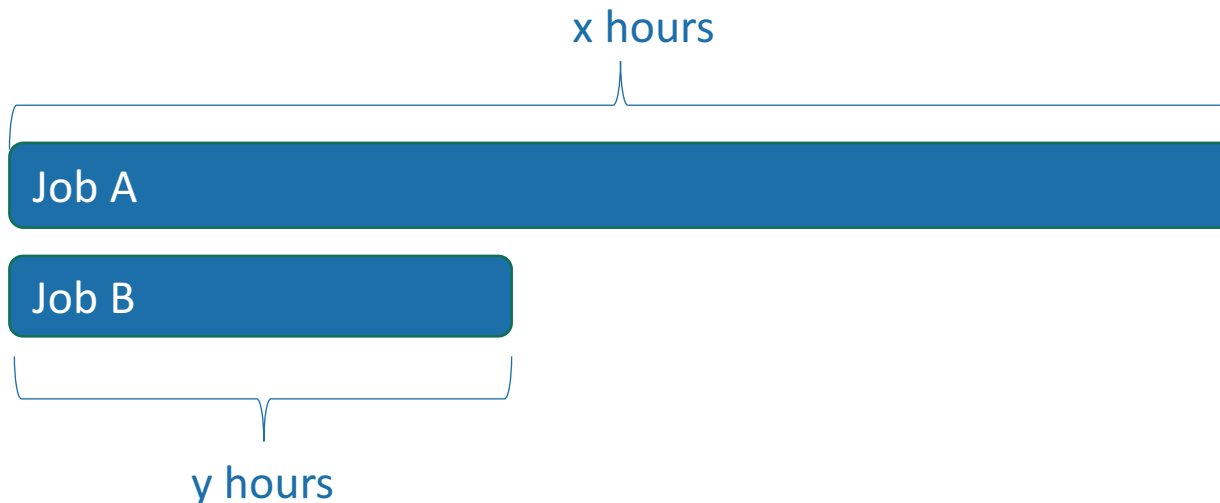
$$xz + (x + y)w \leq yw + (x + y)z$$

$$xz + xw + yw \leq yw + xz + yz$$

$$wx \leq yz$$

$$\frac{w}{y} \leq \frac{z}{x}$$

- Of these two jobs, which should we do first?



Cost: z units per hour until it's done.

Cost: w units per hour until it's done.

- $\text{Cost}(\text{A then B}) = x \cdot z + (x + y) \cdot w$
- $\text{Cost}(\text{B then A}) = y \cdot w + (x + y) \cdot z$

What matters is the ratio:

cost of delay
time it takes

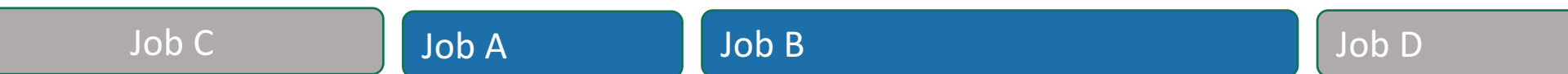
Do the job with the biggest ratio first.

Lemma

- Given jobs so that **Job i** takes time t_i with cost c_i ,
- There is an optimal schedule so that the first job is the one that maximizes the ratio c_i/t_i

- **Proof:**

- Say Job B maximizes this ratio, and it's not first:



$$c_A/t_A \geq c_B/t_B$$

- Switch A and B! Nothing else will change, and we showed on the previous slide that the cost won't increase.



- Repeat until B is first.

Greedy Scheduling Solution

- **scheduleJobs(JOBS)**:
 - Sort JOBS by the ratio:
 - $r_i = \frac{c_i}{t_i} = \frac{\text{cost of delaying job } i}{\text{time job } i \text{ takes to complete}}$
 - Say that **sorted_JOBS[i]** is the job with the i 'th biggest r_i
 - Return **sorted_JOBS**

The running time is $O(n \log(n))$



Formally, we'd use induction

to prove this works

- **Inductive hypothesis:**

- There is an optimal ordering so that the first t jobs are **sorted_JOBS**[1.. t].

- **Base case:**

- When $t=0$, this reads: “There is an optimal ordering so that the first 0 jobs are []”
- That's true.

- **Inductive Step:**

- Boils down to: there is an optimal ordering on **sorted_JOBS**[$t+1..n$] so that **sorted_JOBS**[t] is first.
- This follows from the Lemma.

- **Conclusion:**

- When $t=n$, this reads: “There is an optimal ordering so that the first n jobs are **sorted_JOBS**.”
- aka, what we returned is an optimal ordering.

What have we learned?

- We saw that **scheduling** is another example where a **greedy algorithm** works.
- This followed the same outline as the previous example:
 - Identify **optimal substructure**:



- Find a way to make “safe” choices that **won’t rule out an optimal solution**.
 - **smallest ratios first.**

One more example

Huffman coding

- everyday english sentence

- 01100101 01110110 01100101 01110010 01111001 01100100 01100001
01111001 00100000 01100101 01101110 01100111 01101100 01101001
01110011 01101000 00100000 01110011 01100101 01101110 01110100
01100101 01101110 01100011 01100101

- qwertyui_opasdfg+hjklzxcv

- 01110001 01110111 01100101 01110010 01110100 01111001 01110101
01101001 01011111 01101111 01110000 01100001 01110011 01100100
01100110 01100111 00101011 01101000 01101010 01101011 01101100
01111010 01111000 01100011 01110110

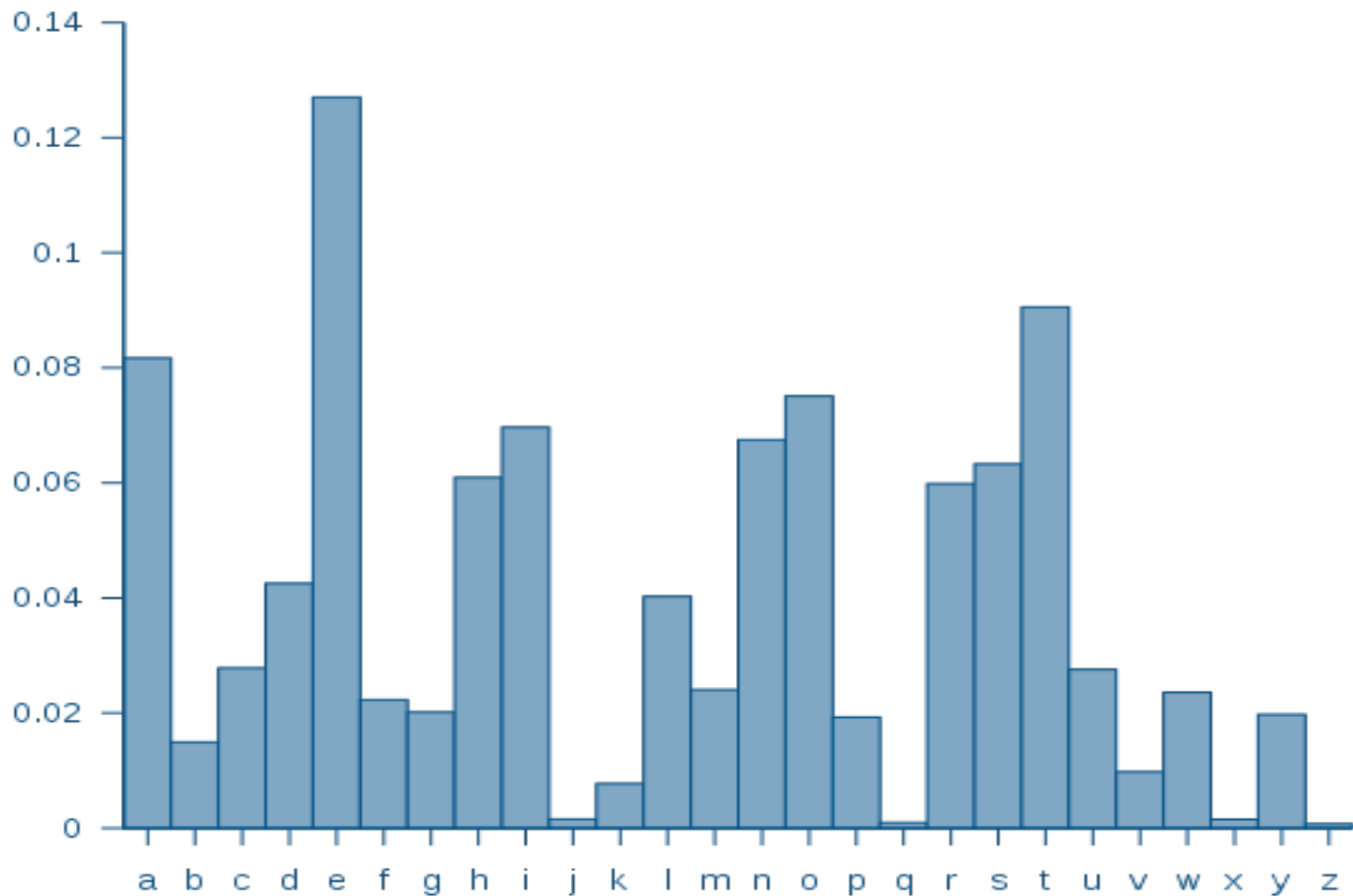
One more example

Huffman coding

ASCII is pretty wasteful. If **e** shows up so often, we should have a more parsimonious way of representing it!

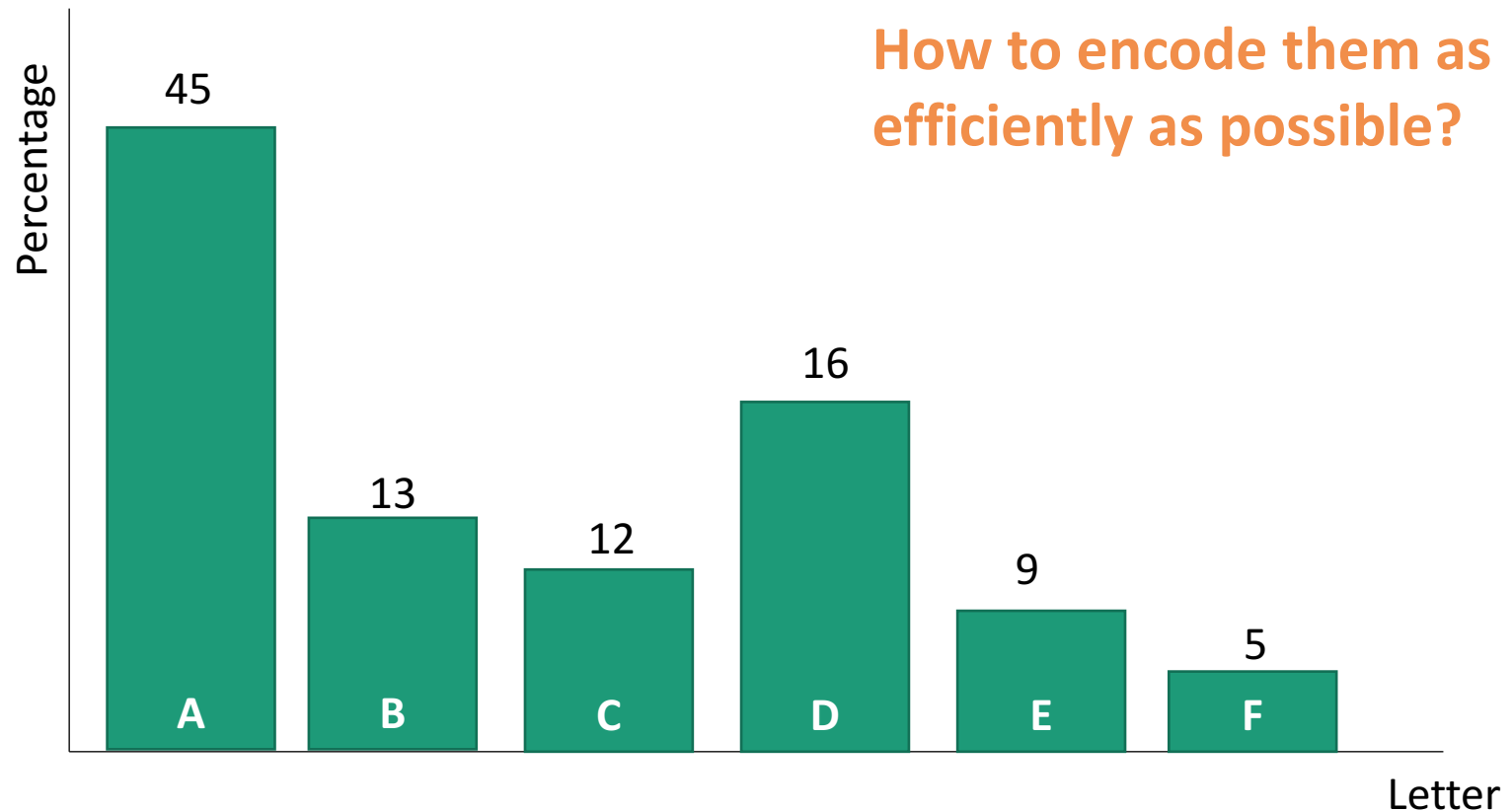
- **e**veryday **e**nglish **s**entence
- **01100101** 01110110 **01100101** 01110010 01111001 01100100 01100001
01111001 00100000 **01100101** 01101110 01100111 01101100 01101001
01110011 01101000 00100000 01110011 **01100101** 01101110 01110100
01100101 01101110 01100011 **01100101**
- qwertyui_opasdfg+hjklzxcv
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101
01101001 01011111 01101111 01110000 01100001 01110011 01100100
01100110 01100111 00101011 01101000 01101010 01101011 01101100
01111010 01111000 01100011 01110110

Suppose we have some
distribution on characters



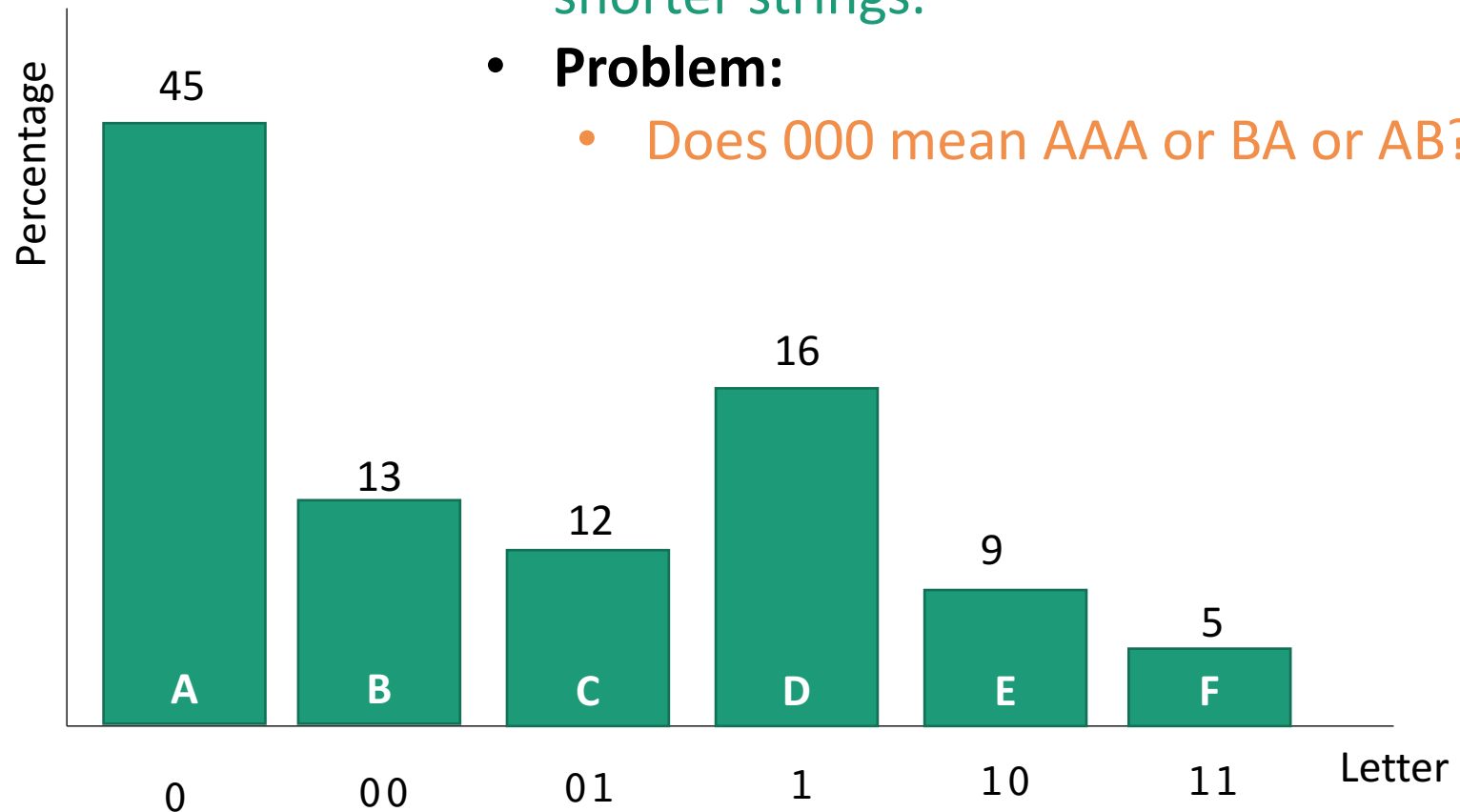
Suppose we have some distribution on characters

For simplicity,
let's go with this
made-up example



Try 1

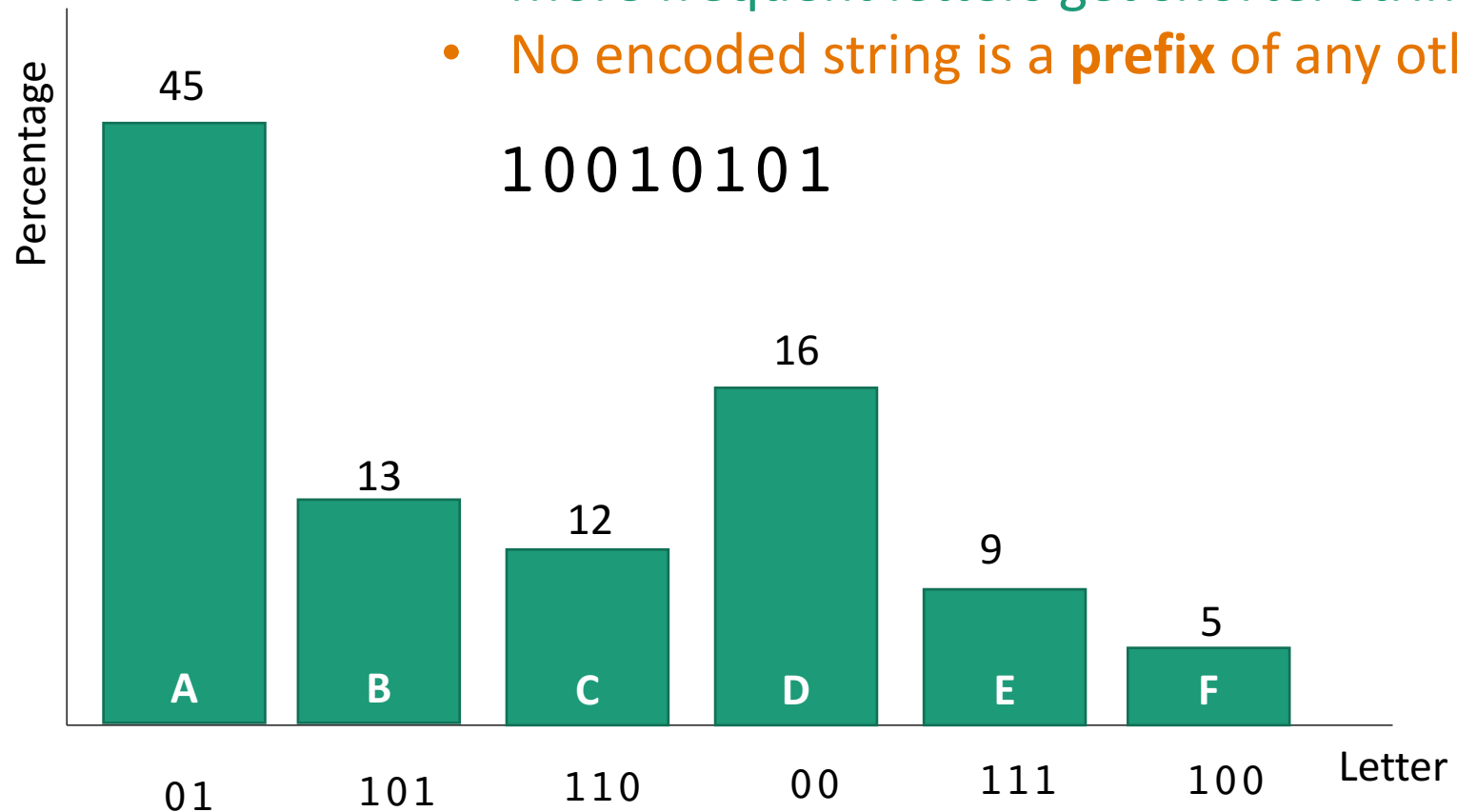
- Every letter is assigned a **binary string** of one or two bits.
- The more frequent letters get the shorter strings.
- **Problem:**
 - Does 000 mean AAA or BA or AB?



Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

Try 2: prefix-free coding

- Every letter is assigned a **binary string**.
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.

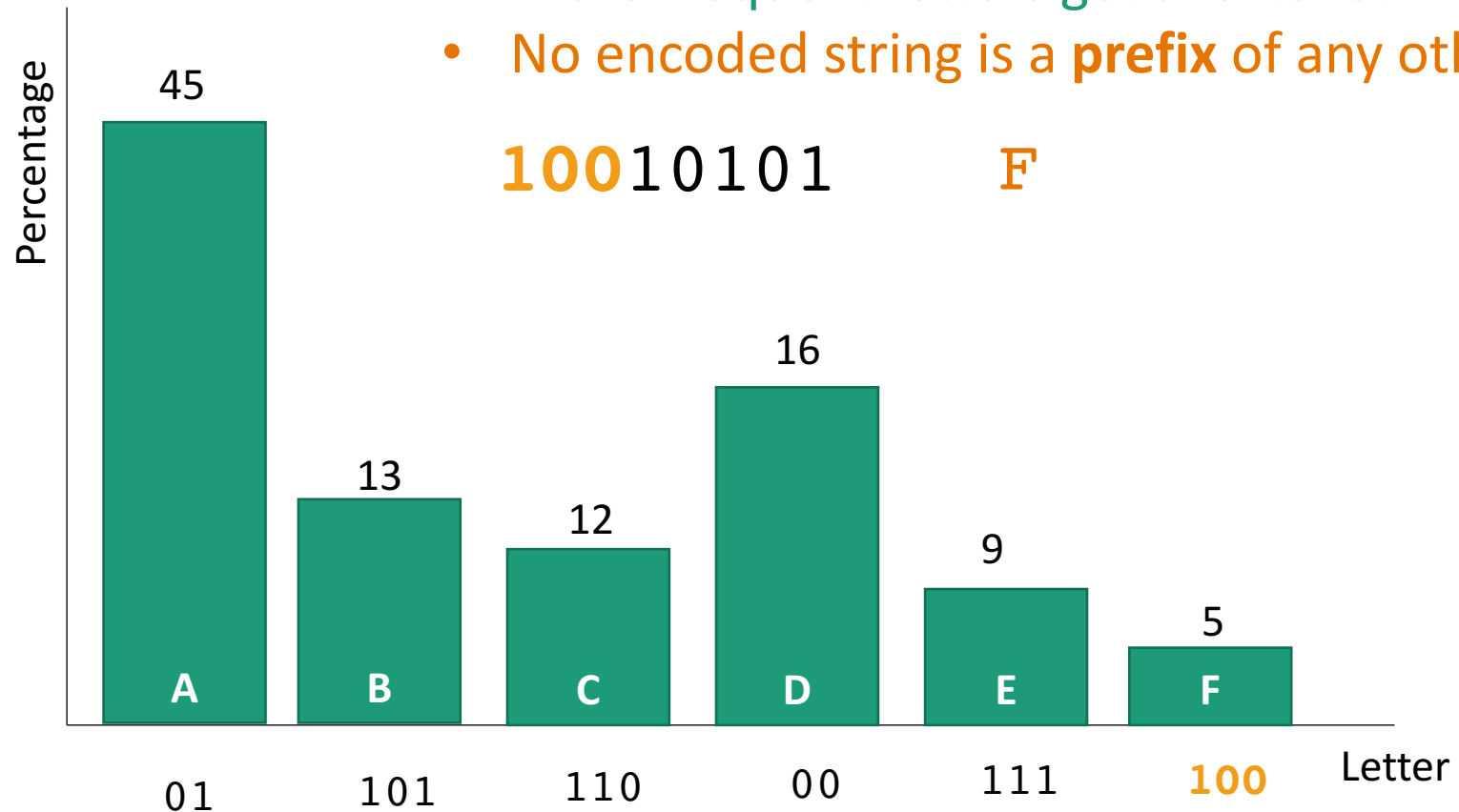


10010101

Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

Try 2: prefix-free coding

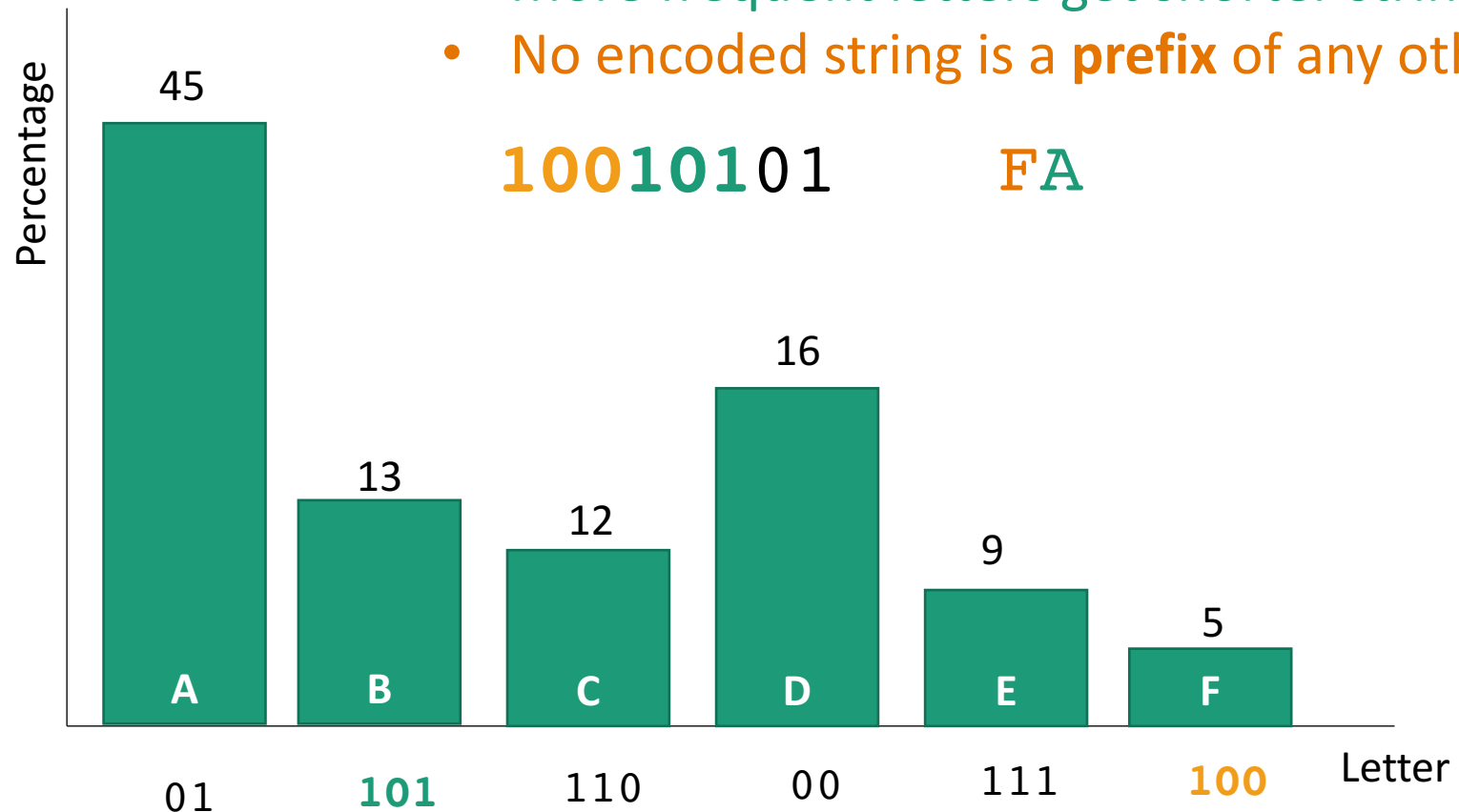
- Every letter is assigned a **binary string**.
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.



Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

Try 2: prefix-free coding

- Every letter is assigned a **binary string**.
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.



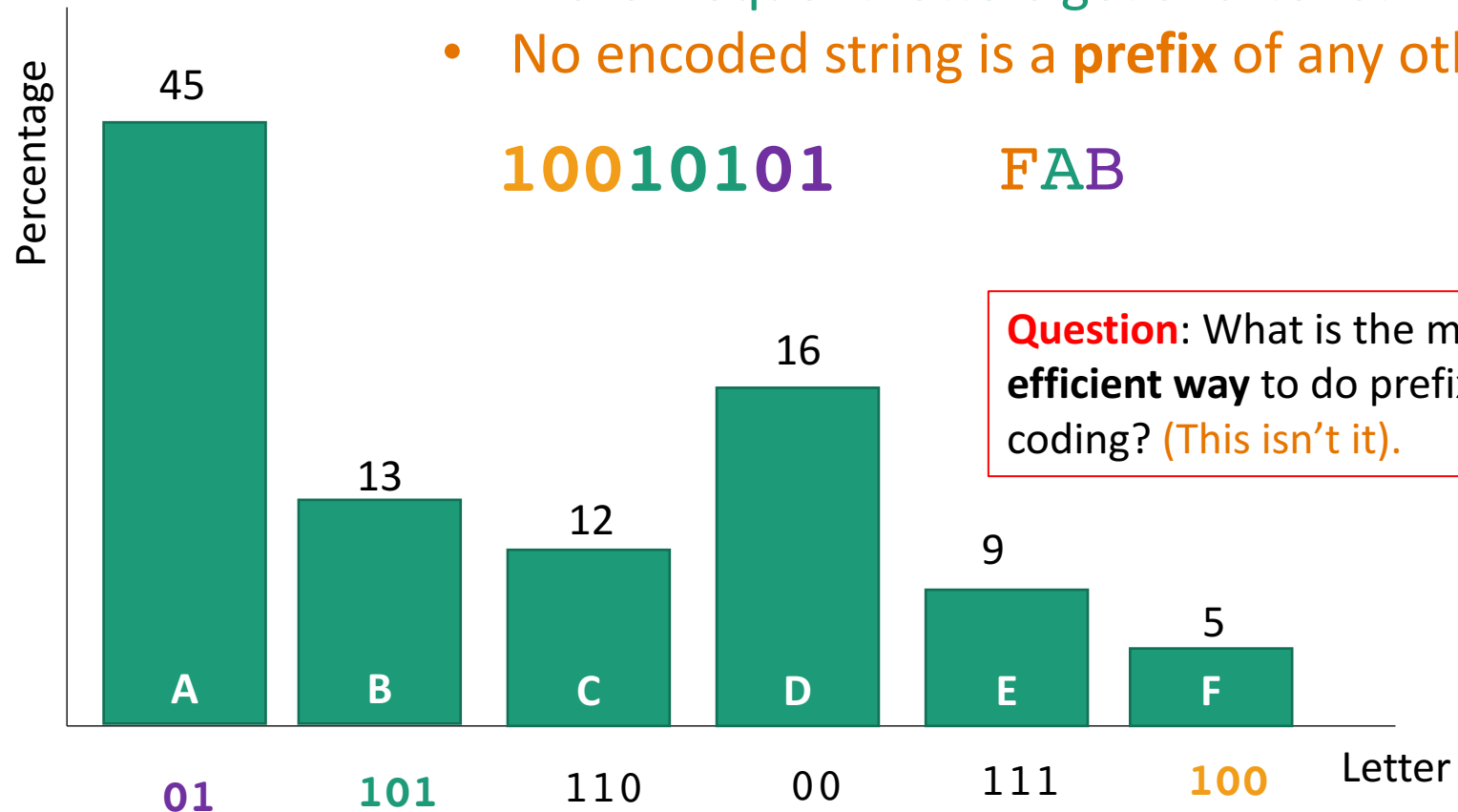
10010101

FA

Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

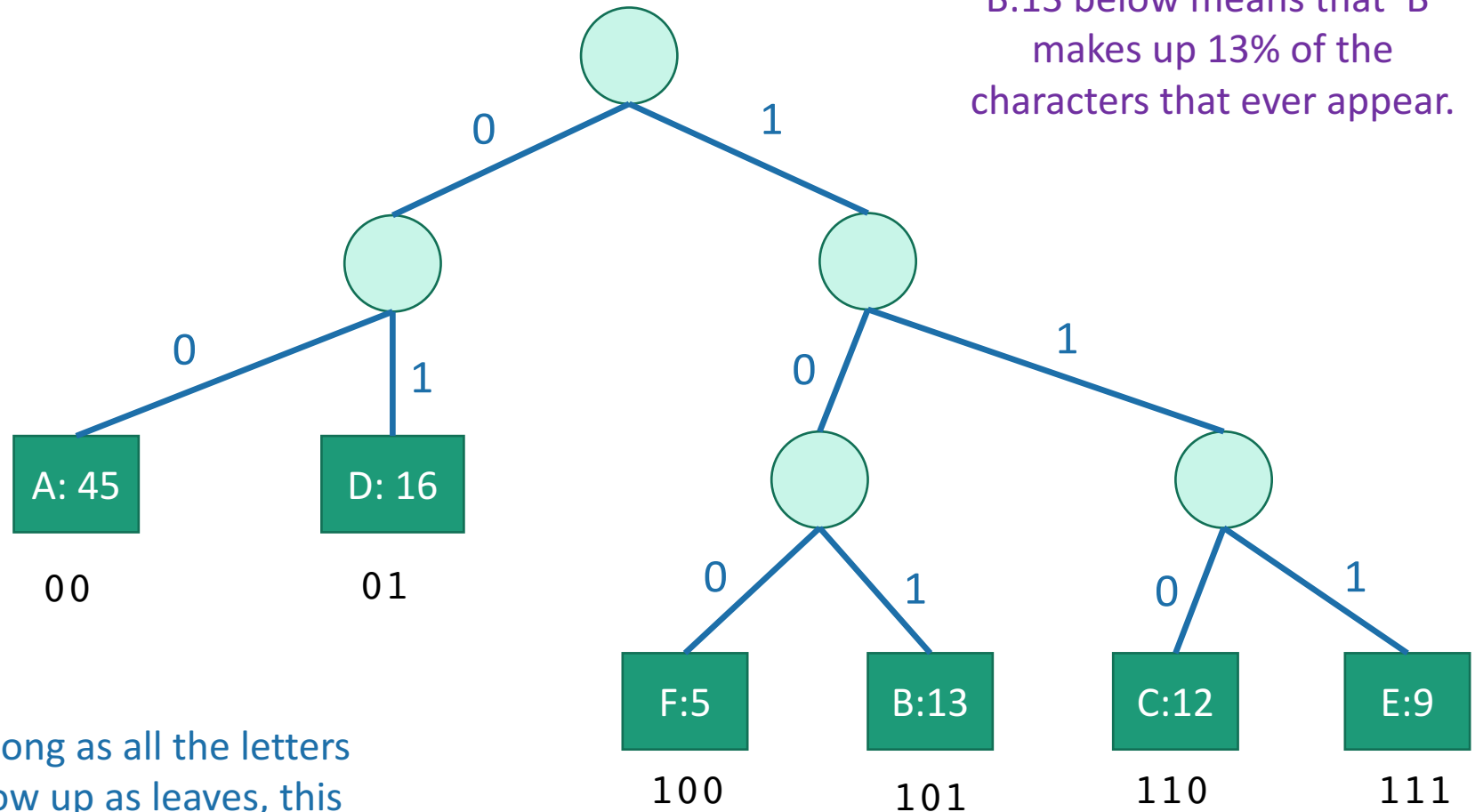
Try 2: prefix-free coding

- Every letter is assigned a **binary string**.
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.



A prefix-free code is a tree

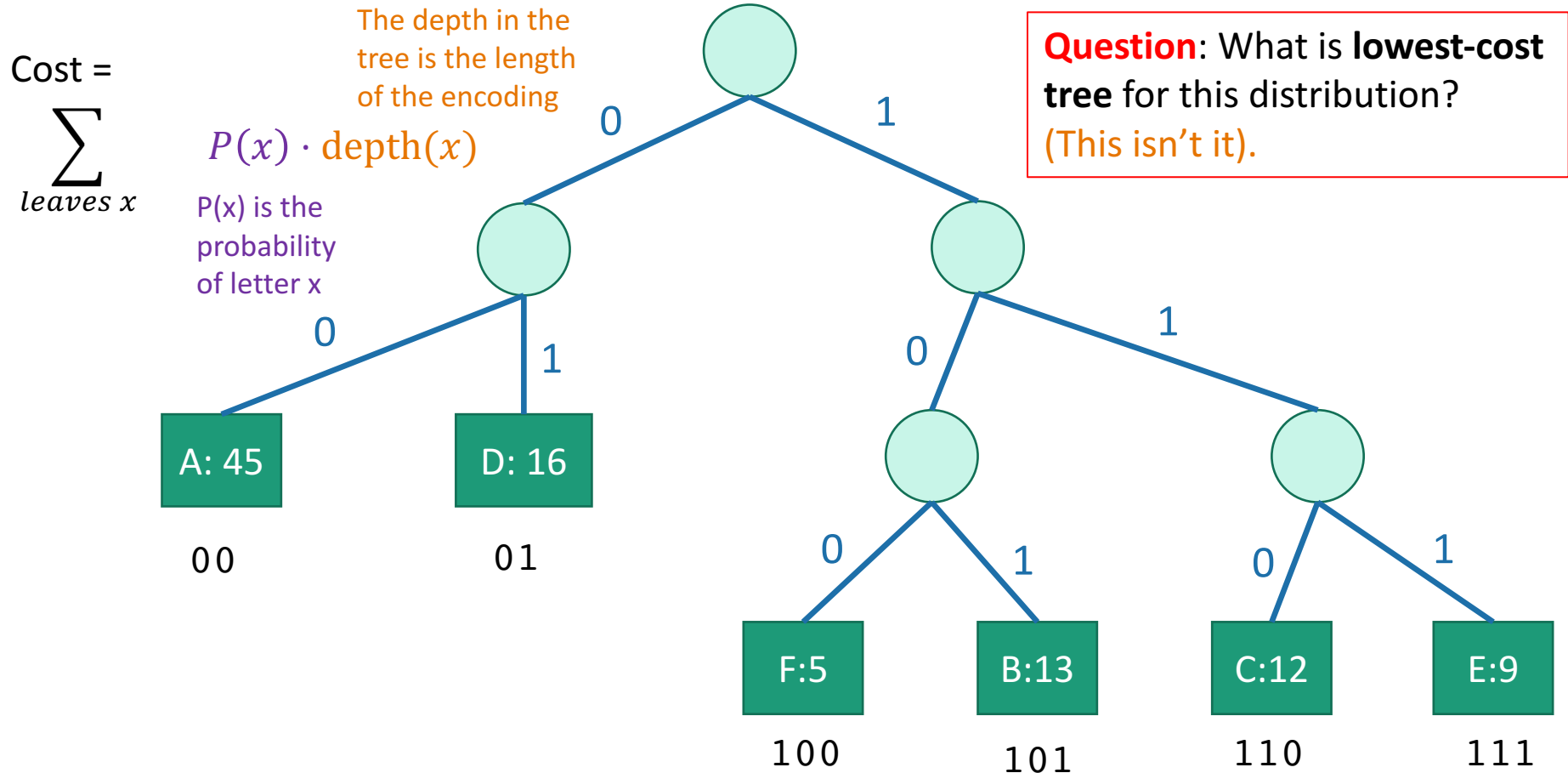
B:13 below means that 'B'
makes up 13% of the
characters that ever appear.



As long as all the letters
show up as leaves, this
code is **prefix-free**.

Some trees are better than others

- Imagine choosing a letter at random from the language.
 - Not uniform, but according to our histogram!
- The **cost of a tree** is the expected length of the encoding of that letter.

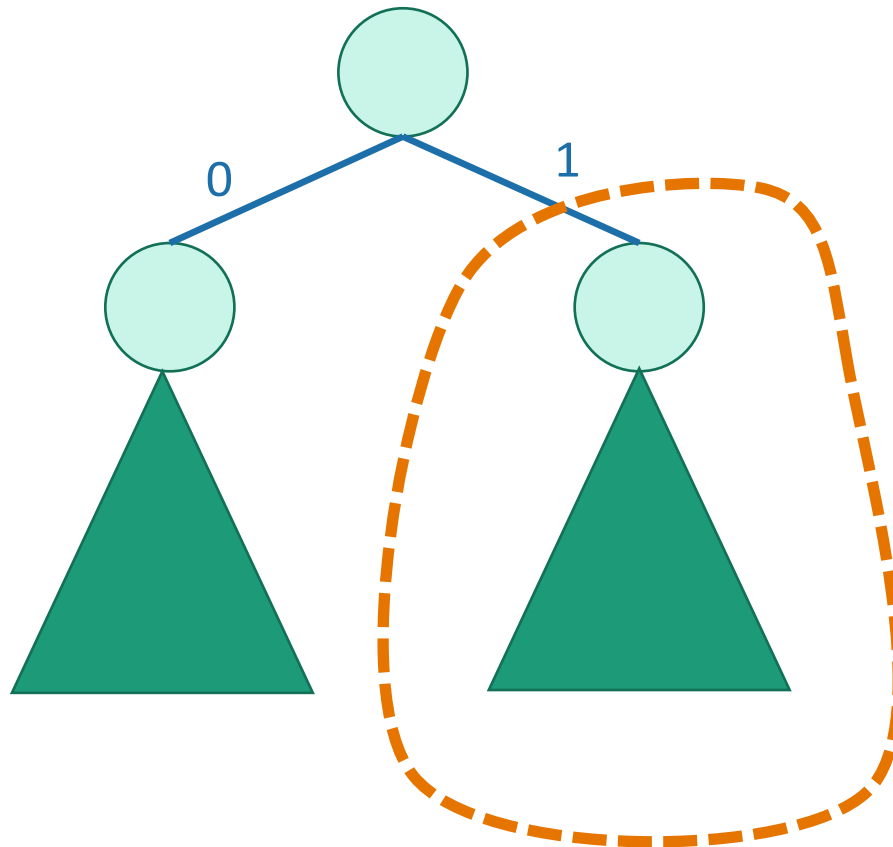


Expected cost of encoding a letter with this tree:

$$2(0.45 + 0.16) + 3(0.05 + 0.13 + 0.12 + 0.09) = 2.39$$

Optimal sub-structure

- Suppose this is an optimal tree:

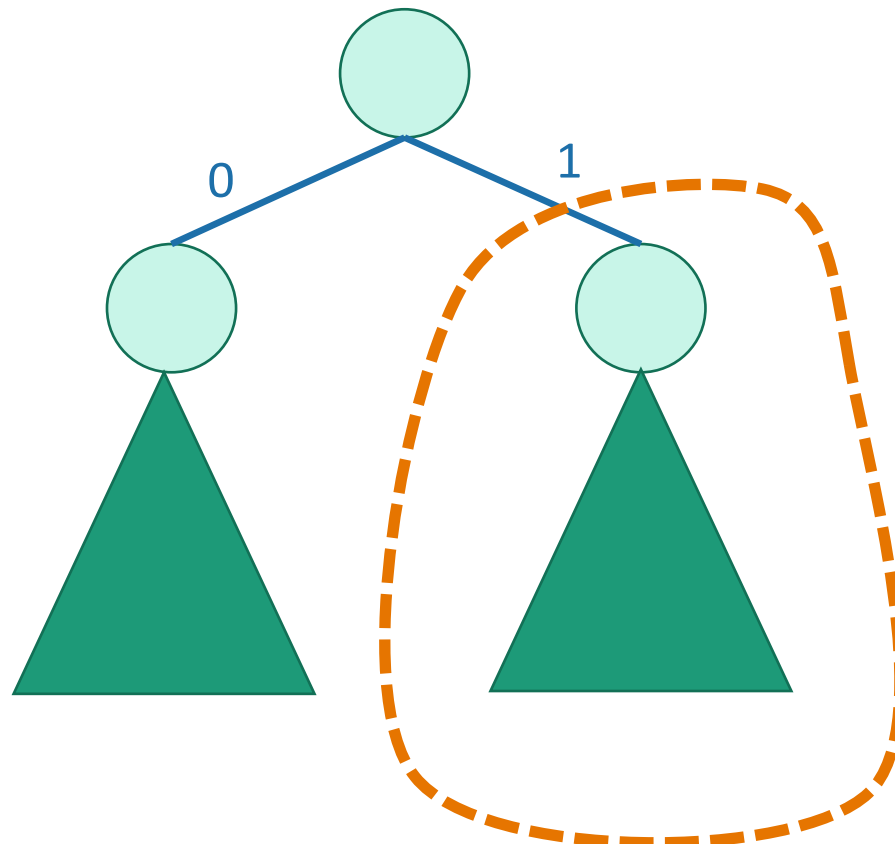


Then this is an optimal tree on fewer letters.

Otherwise, we could change this sub-tree and end up with a better overall tree.

In order to design a greedy algorithm

- Think about what letters belong in this sub-problem...



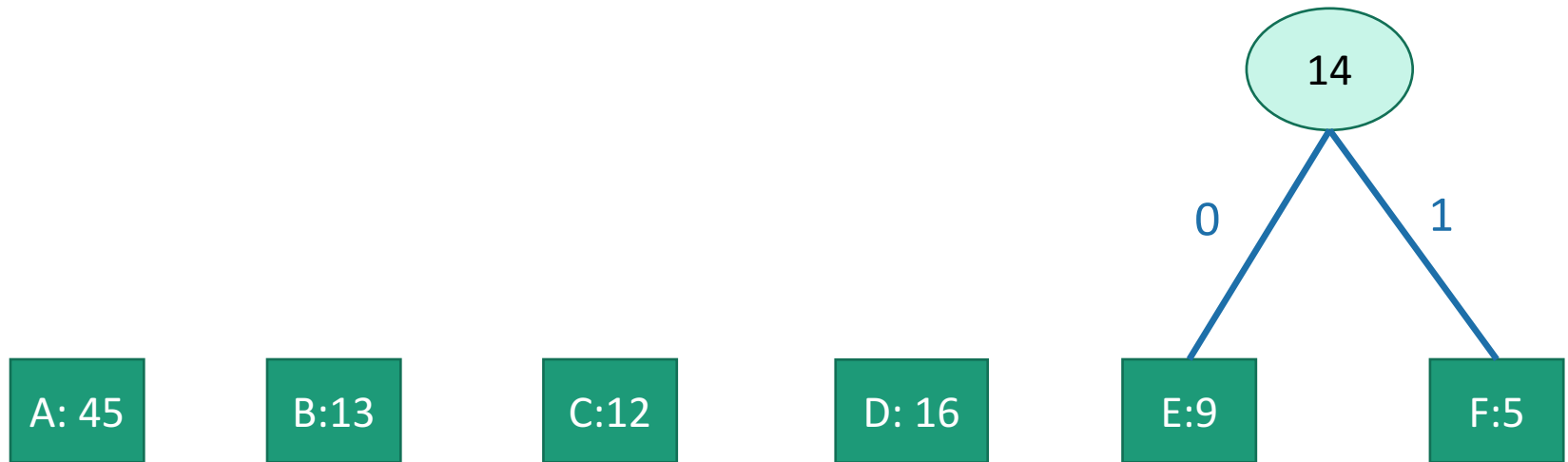
What's a **safe choice** to make for these lower sub-trees?

Infrequent elements!

We want them as low down as possible.

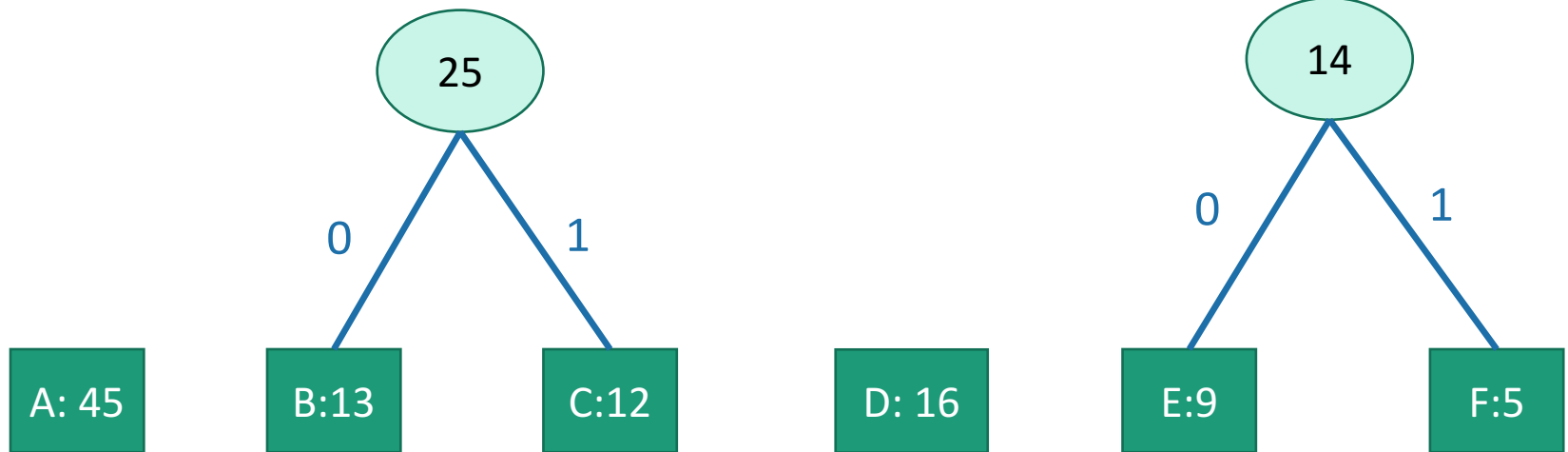
Solution

greedily build subtrees, starting with the infrequent letters



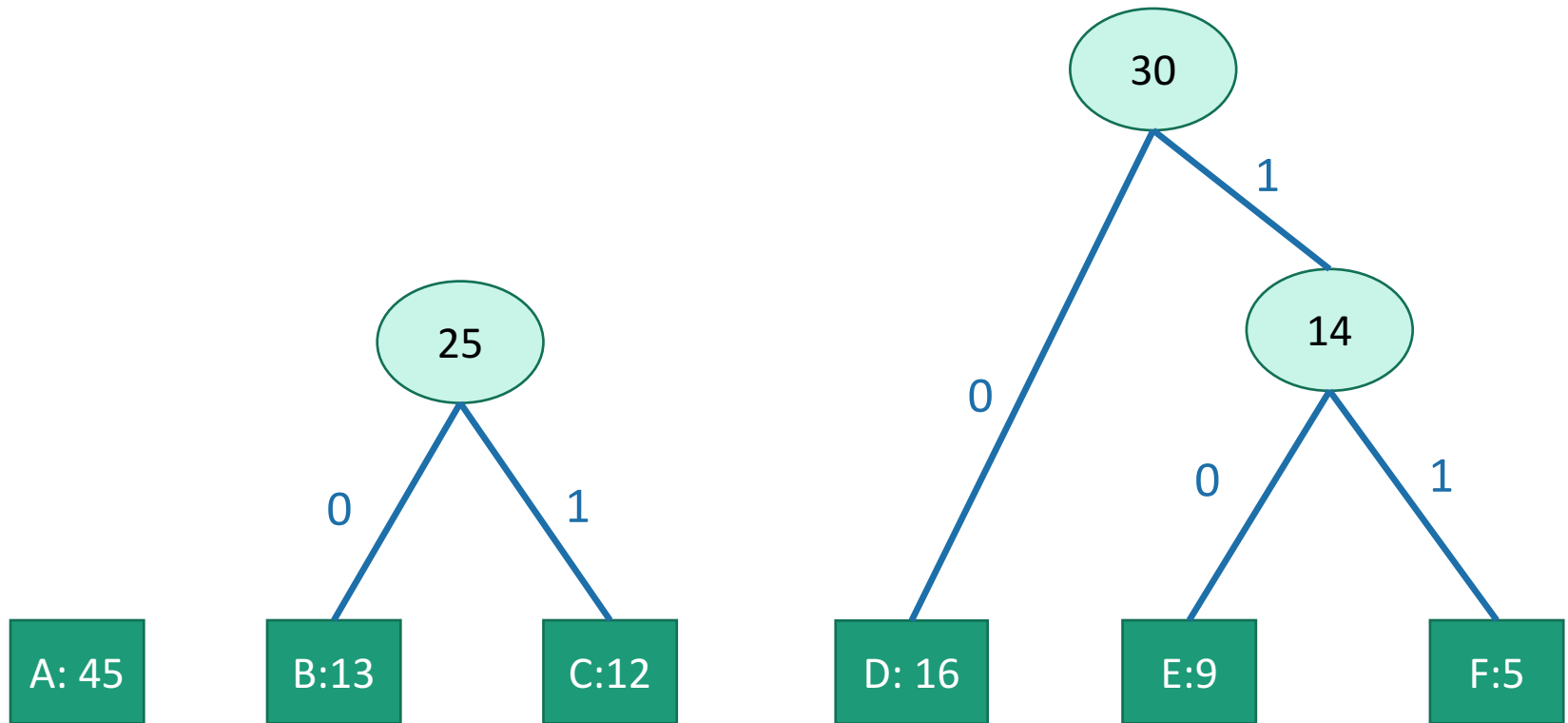
Solution

greedily build subtrees, starting with the infrequent letters



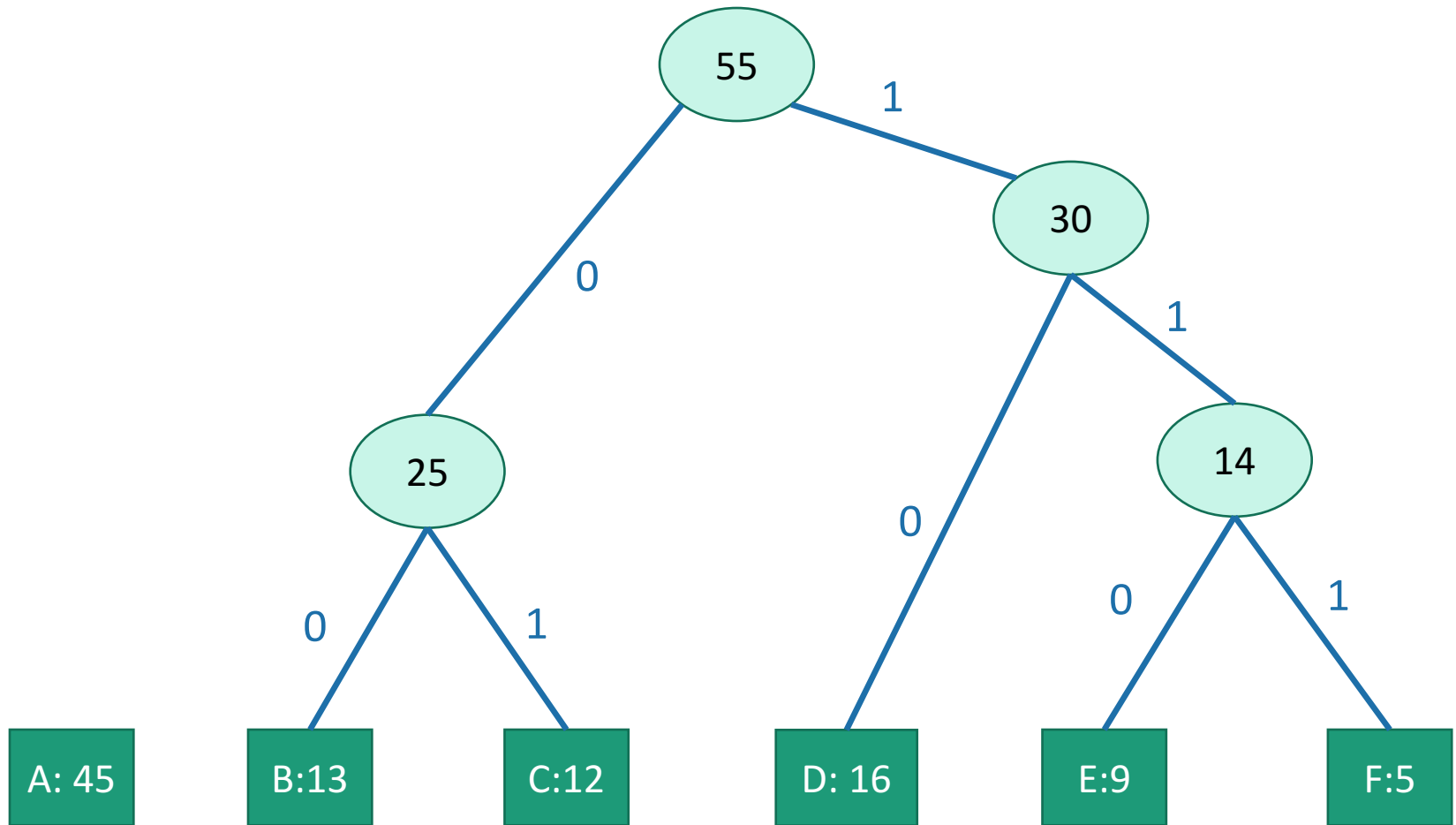
Solution

greedily build subtrees, starting with the infrequent letters



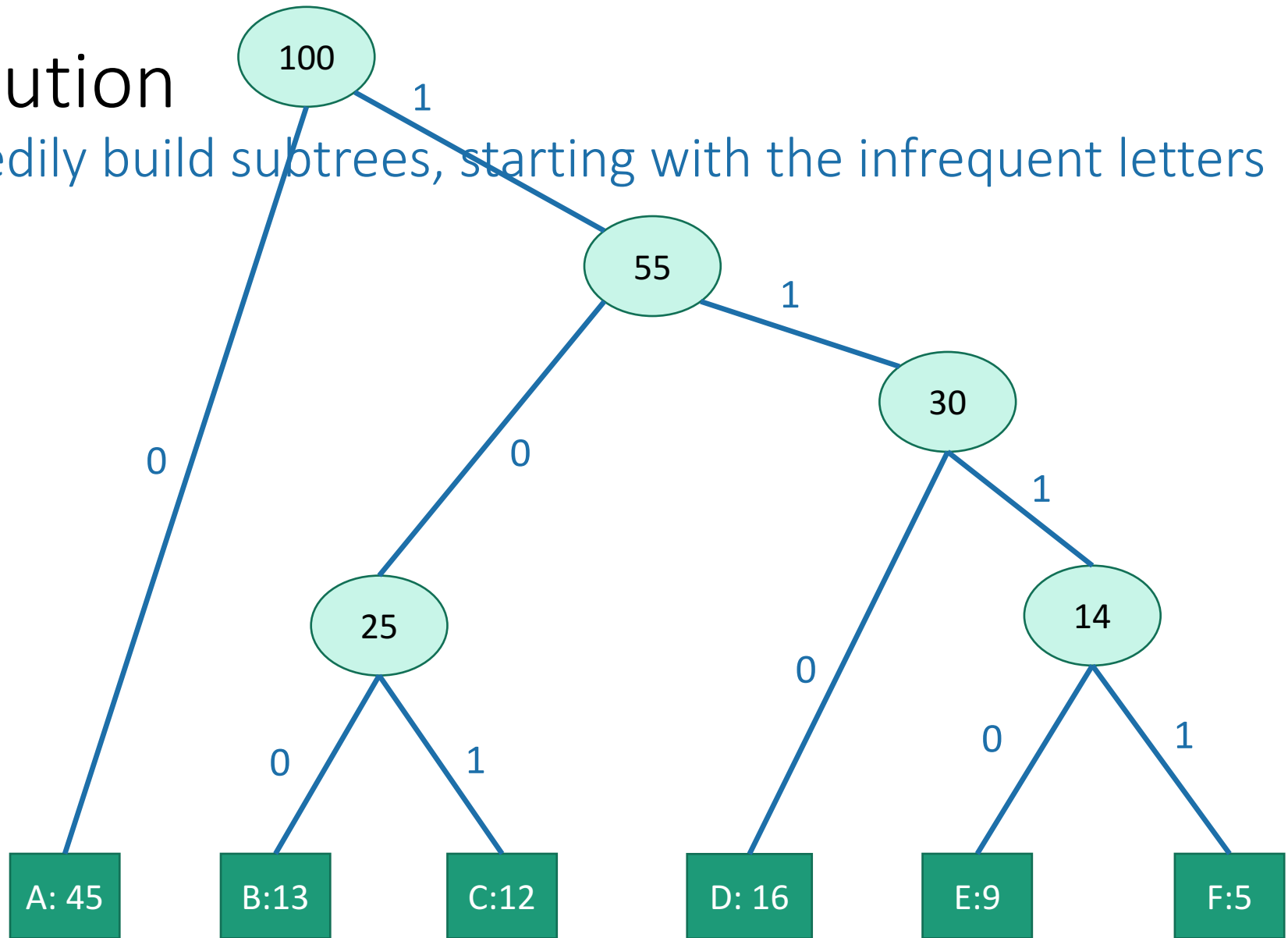
Solution

greedily build subtrees, starting with the infrequent letters



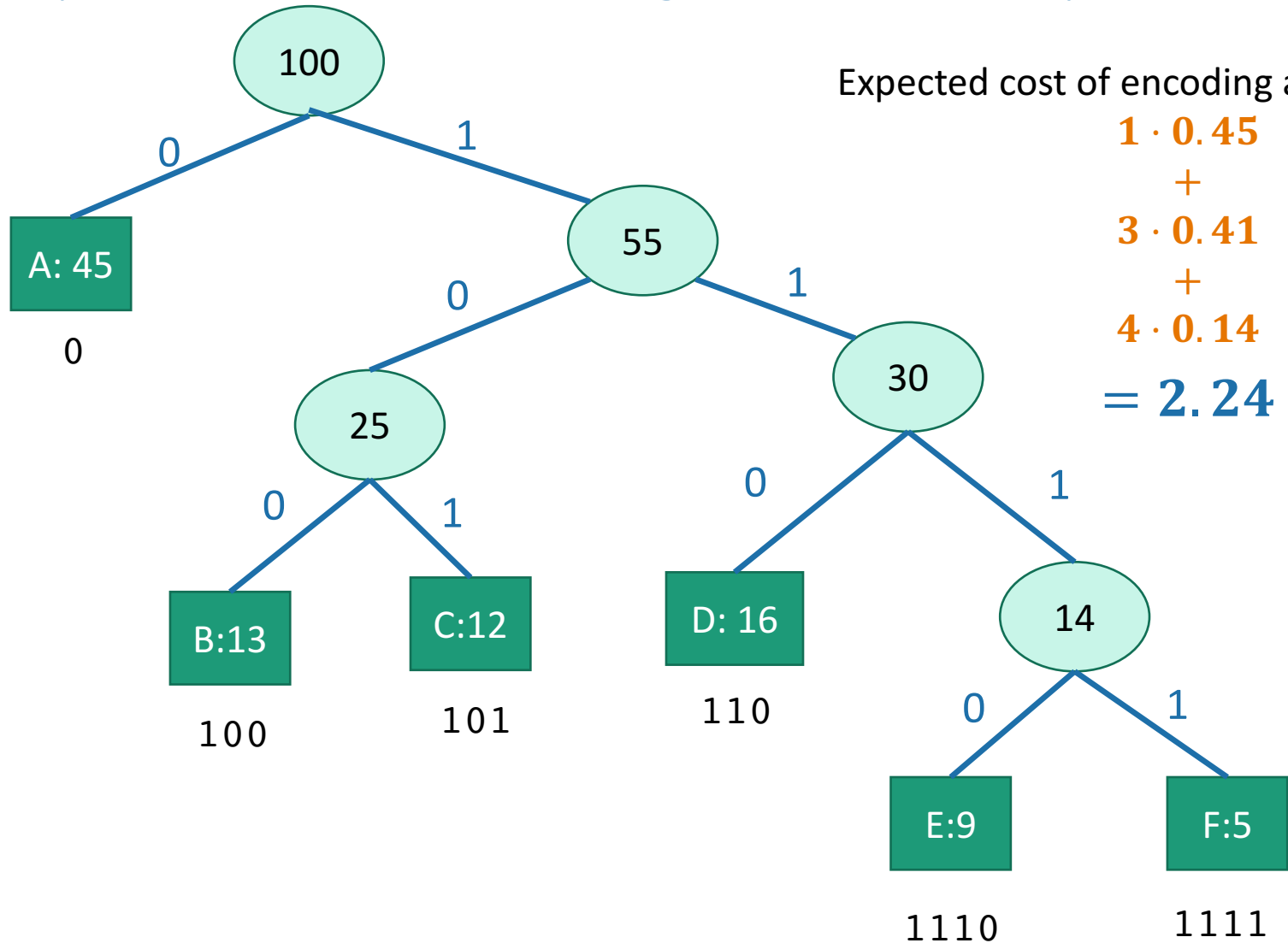
Solution

greedily build subtrees, starting with the infrequent letters



Solution

greedily build subtrees, starting with the infrequent letters



What exactly was the algorithm?

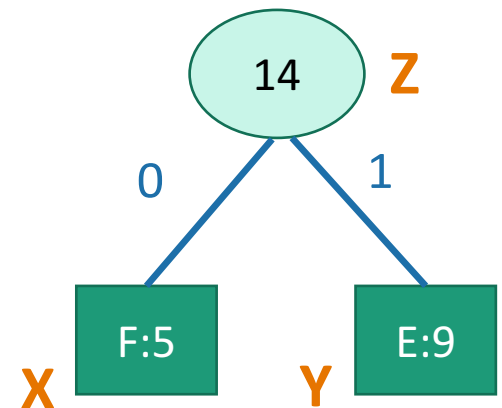
- Create a node like **D: 16** for each letter/frequency
 - The key is the frequency (16 in this case)
- Let **CURRENT** be the list of all these nodes.
- **while** len(**CURRENT**) > 1:
 - **X** and **Y** ← the nodes in **CURRENT** with the smallest keys.
 - Create a new node **Z** with **Z.key = X.key + Y.key**
 - Set **Z.left = X, Z.right = Y**
 - Add **Z** to **CURRENT** and remove **X** and **Y**
- return **CURRENT**[0]

A: 45

B: 13

C: 12

D: 16



Proof strategy

just like before

- Show that at each step, the choices we are making **won't rule out** an optimal solution.
- **Lemma:**
 - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.

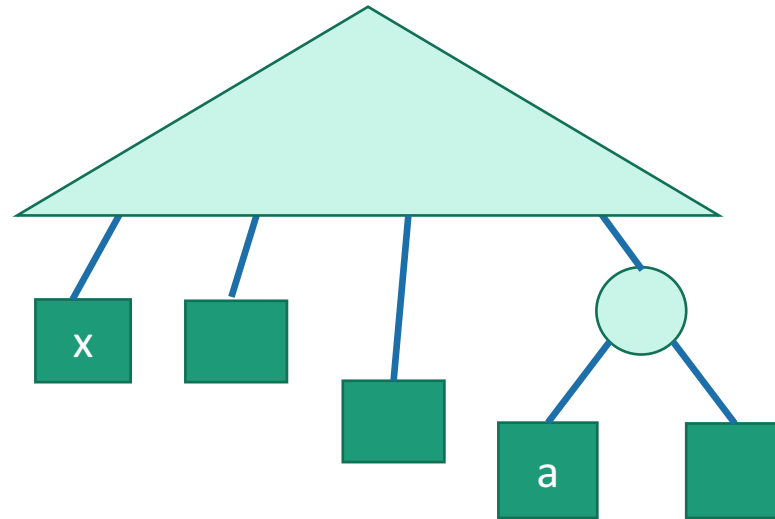


Lemma

proof idea

If x and y are the two least-frequent letters, there is an optimal subtree where x and y are siblings.

- Say that an optimal tree looks like this:



Lowest-level sibling nodes: at least one of them is neither x nor y

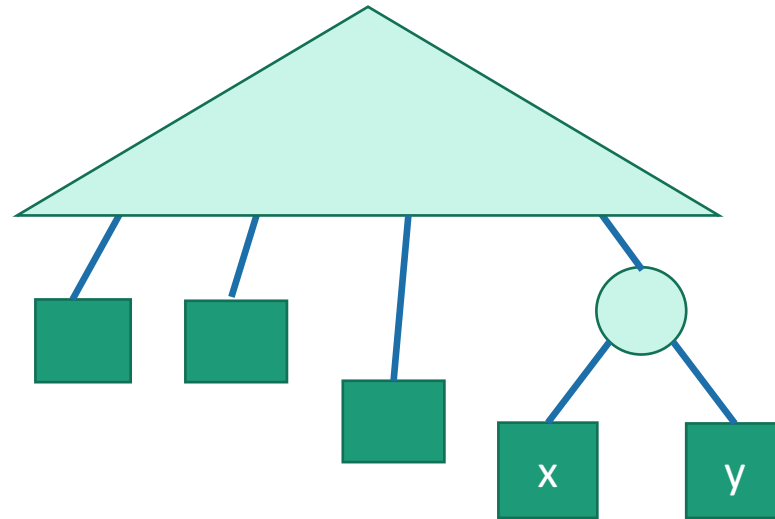
- What happens to the cost if we swap x for a ?
 - the cost can't increase; a was more frequent than x , and we just made its encoding shorter.
- Repeat this logic until we get an optimal tree with x and y as siblings.

Lemma

proof idea

If x and y are the two least-frequent letters, there is an optimal subtree where x and y are siblings.

- Say that an optimal tree looks like this:



Lowest-level sibling nodes: at least one of them is neither x nor y

- What happens to the cost if we swap x for a ?
 - the cost can't increase; a was more frequent than x , and we just made its encoding shorter.
- Repeat this logic until we get an optimal tree with x and y as siblings.

Proof strategy

just like last time

- Show that at each step, the choices we are making **won't rule out** an optimal solution.
- Lemma:
 - Suppose that x and y are the two least-frequent letters.
Then there is an optimal tree where x and y are siblings.

Actually that's not quite enough...



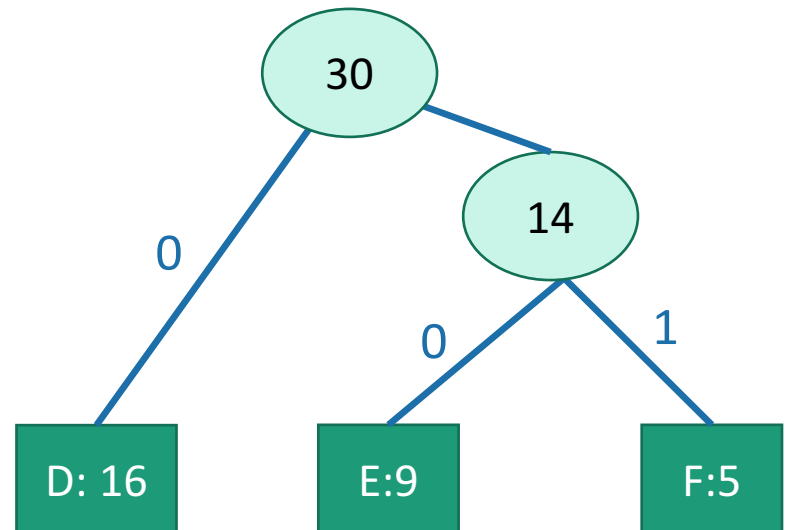
Proof strategy

just like last time

Our argument before just showed that we made the right choice at the first step, when everything was a leaf. What about once we start grouping stuff?

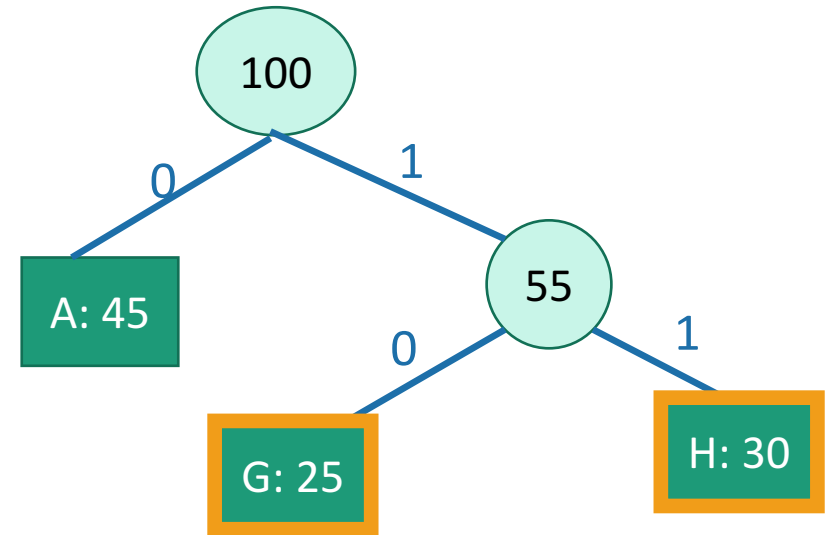
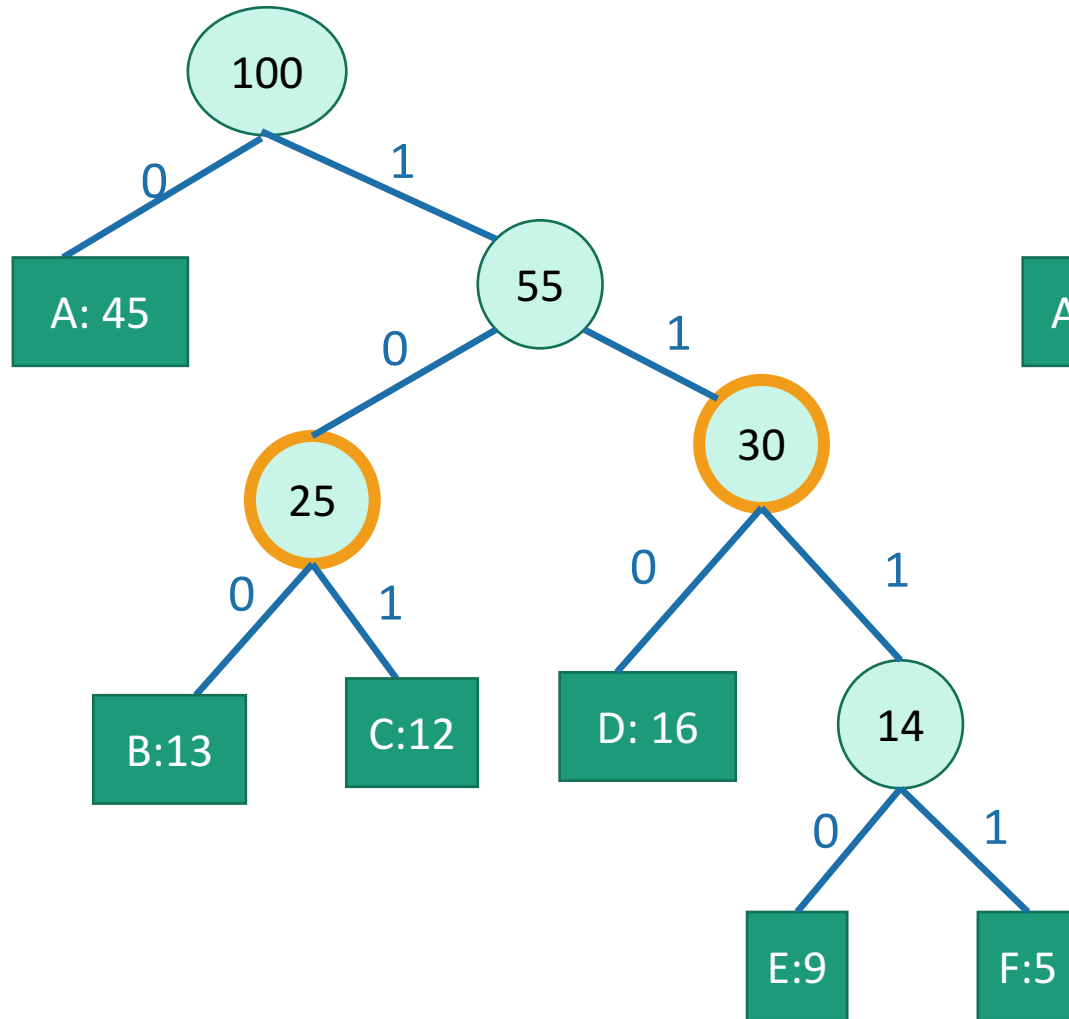
- Show that at each step, the choices we are making **won't rule out** an optimal solution.
- Lemma:
 - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.

Actually that's not quite enough...



Lemma 2

this distinction doesn't really matter



The first thing is an optimal tree on $\{A,B,C,D,E,F\}$

if and only if

the second thing is an optimal tree on $\{A,G,H\}$

Lemma 2

this distinction doesn't really matter

- For a proof:
 - See CLRS, Lemma 16.3
 - Rigorous although presented in a slightly different way
 - See Lecture Notes 14
 - A bit sketchier, but presented in the same way as here
 - Prove it yourself!
 - This is the best!

Getting all the details
isn't that important, but
you should convince
yourself that this is true.



Ollie the over-achieving ostrich

Together

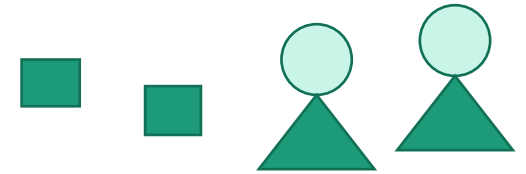
- Lemma 1:
 - Suppose that x and y are the two least-frequent letters.
Then there is an optimal tree where x and y are siblings.
- Lemma 2:
 - We may as well imagine that CURRENT contains only leaves.
- These imply:
 - At each step, our choice doesn't rule out an optimal tree.

Formally, we'd use induction

After the t 'th step, we've got a bunch of current sub-trees:

- Inductive hypothesis:

- after the t 'th step,
 - there is an optimal tree containing the current subtrees as “leaves”



- Base case:

- after the 0'th step,
 - there is an optimal tree containing all the characters.

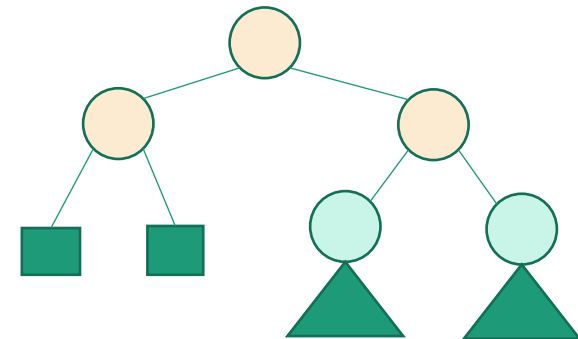
*Inductive hyp. asserts
that our subtrees can be
assembled into an
optimal tree:*

- Inductive step:

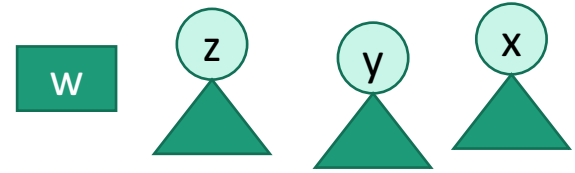
- **TO DO**

- Conclusion:

- after the last step,
 - there is an optimal tree containing this whole tree as a subtree.
- aka,
 - after the last step the tree we've constructed is optimal.

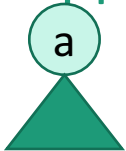


We've got a bunch of current sub-trees:



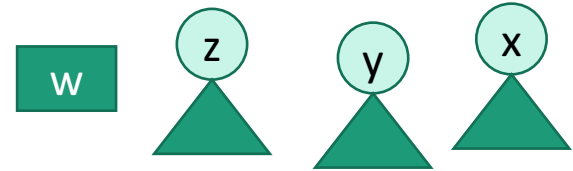
say that x and y are the two smallest.

Inductive step

- Suppose that the inductive hypothesis holds for t-1
 - After t-1 steps, there is an optimal tree containing all the current sub-trees as “leaves.”
- Want to show:
 - After t steps, there is an optimal tree containing all the current sub-trees as leaves.
- Two ingredients:
 - **Lemma 1:** If x and y are the two least-frequent letters, there is an optimal subtree where x and y are siblings.
 - **Lemma 2:** Suppose that there is an optimal tree containing  as a subtree. Then we may as well replace it with a new letter with frequency 

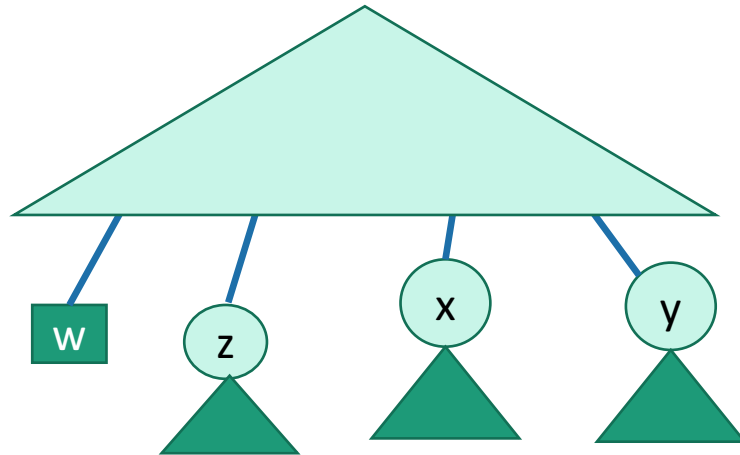
Inductive step

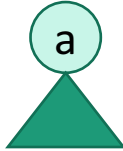
We've got a bunch of current sub-trees:



say that x and y are the two smallest.

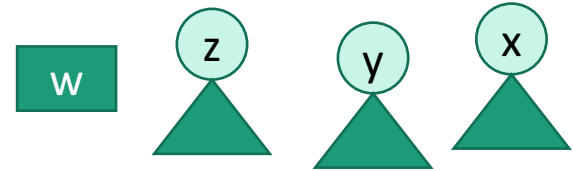
- Suppose that the inductive hypothesis holds for $t-1$
 - After $t-1$ steps, there is an optimal tree containing all the current sub-trees as “leaves”.



- By Lemma 2, may as well treat  as 

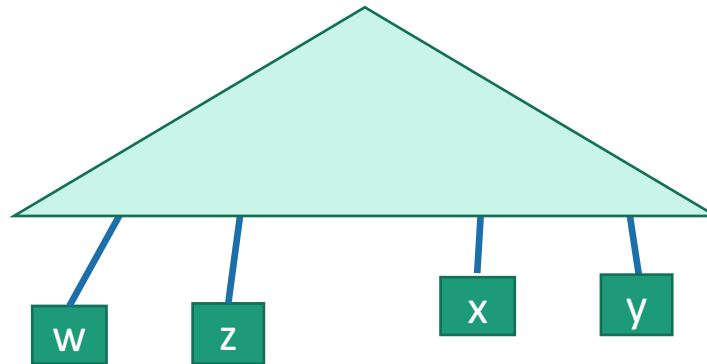
Inductive step

We've got a bunch of current sub-trees:



say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for $t-1$
 - After $t-1$ steps, there is an optimal tree containing all the current sub-trees as “leaves.”



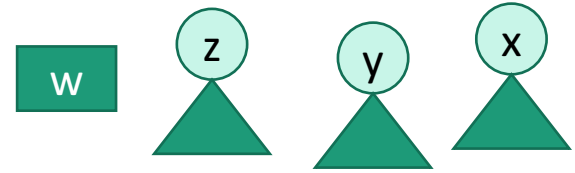
- By Lemma 2, may as well treat



- In particular, optimal trees on this new alphabet correspond to optimal trees on the original alphabet.

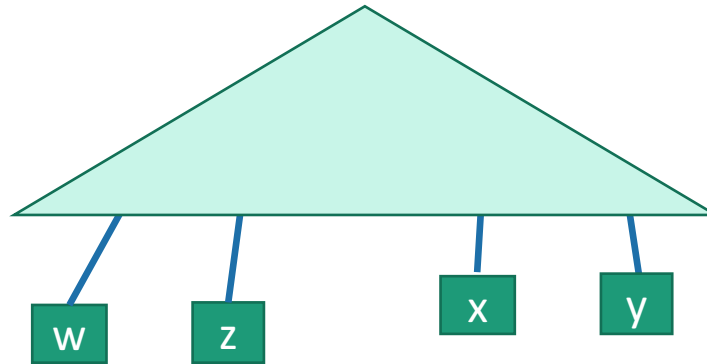
We've got a bunch of current sub-trees:

Inductive step

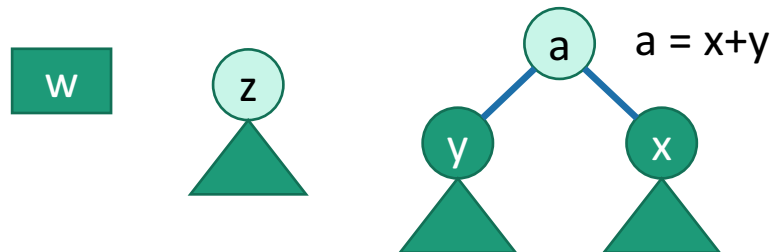


say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for $t-1$
 - After $t-1$ steps, there is an optimal tree containing all the current sub-trees as “leaves.”

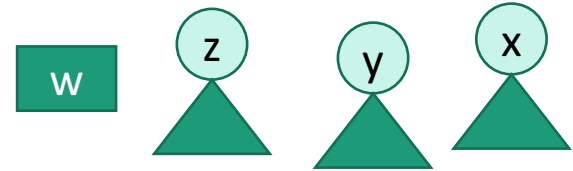


- Our algorithm would do this at level t :



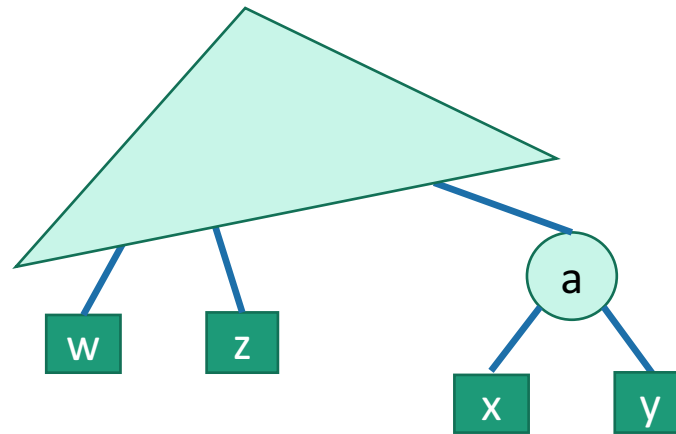
Inductive step

We've got a bunch of current sub-trees:



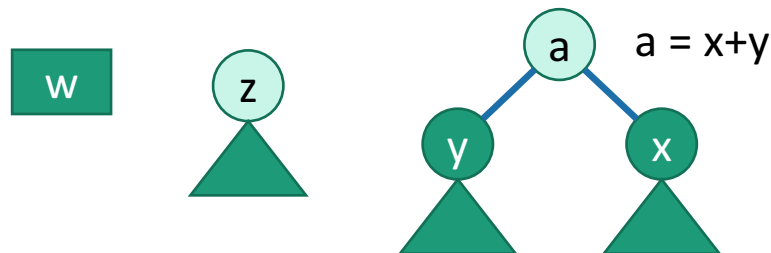
say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for $t-1$
 - After $t-1$ steps, there is an optimal tree containing all the current sub-trees as “leaves.”



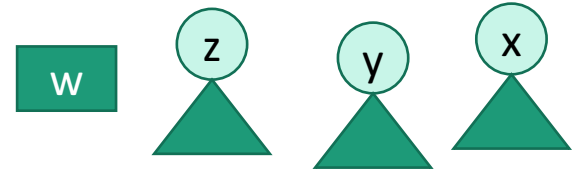
Lemma 1 implies that there's an optimal sub-tree that looks like this; aka, what our algorithm did okay.

- Our algorithm would do this at level t :



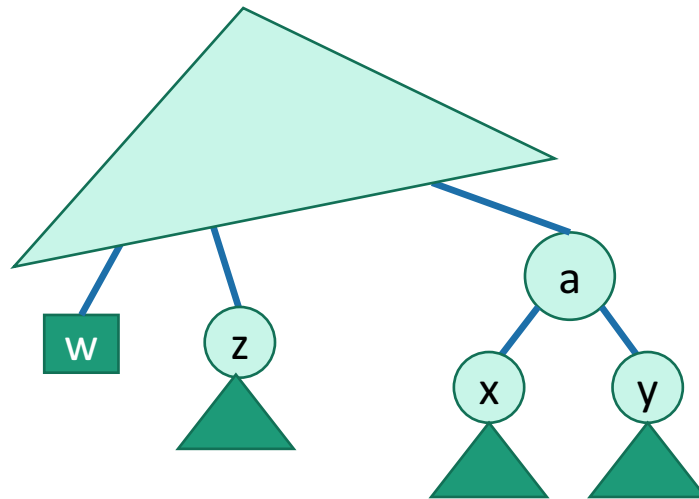
Inductive step

We've got a bunch of current sub-trees:



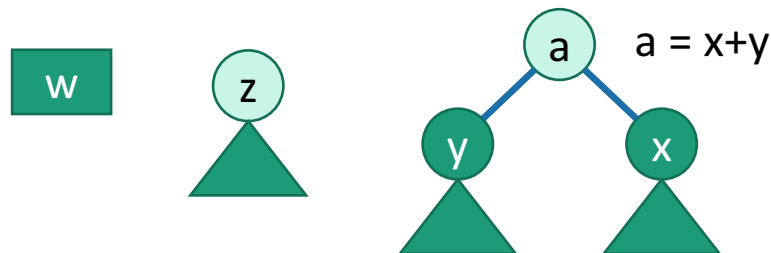
say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for $t-1$
 - After $t-1$ steps, there is an optimal tree containing all the current sub-trees as “leaves.”



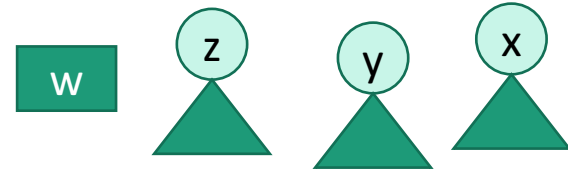
Lemma 2 again says that there's an optimal tree that looks like this

- Our algorithm would do this at level t :



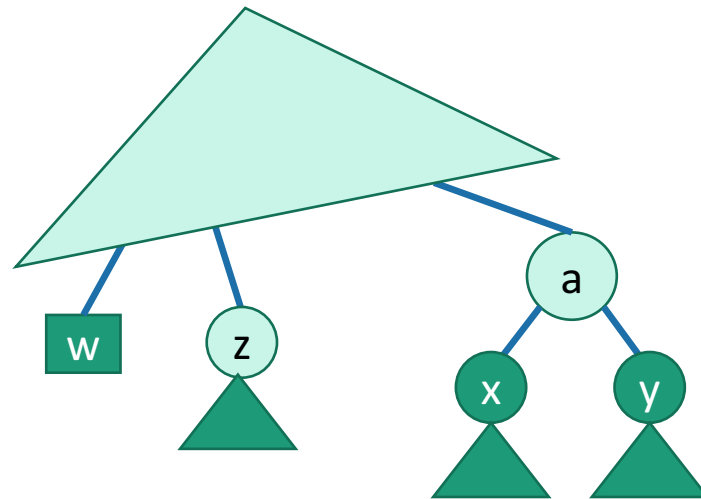
Inductive step

We've got a bunch of current sub-trees:



say that x and y are the two smallest.

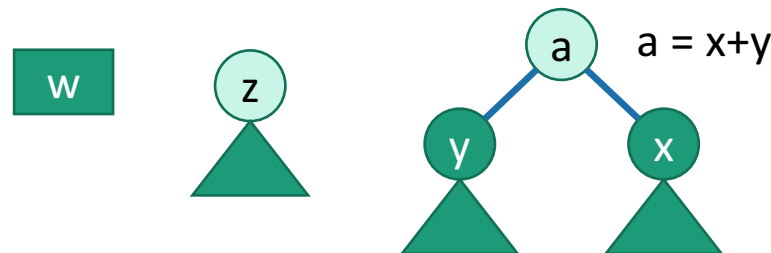
- Suppose that the inductive hypothesis holds for $t-1$
 - After $t-1$ steps, there is an optimal tree containing all the current sub-trees as “leaves.”



Lemma 2 again says that there's an optimal tree that looks like this

aka, there is an optimal tree containing all the level- t sub-trees as “leaves”.

- Our algorithm would do this at level t :



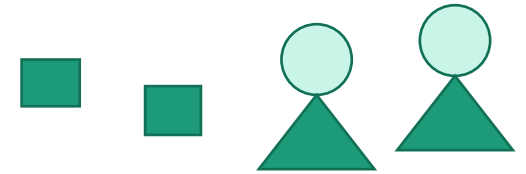
This is what we wanted to show for the inductive step.

Inductive outline:

After the t' th step, we've got a bunch of current sub-trees:

- Inductive hypothesis:

- after the t' th step,
 - there is an optimal tree containing the current subtrees as “leaves”



- Base case:

- after the 0'th step,
 - there is an optimal tree containing all the vertices.

Inductive hyp. asserts that our subtrees can be assembled into an optimal tree:

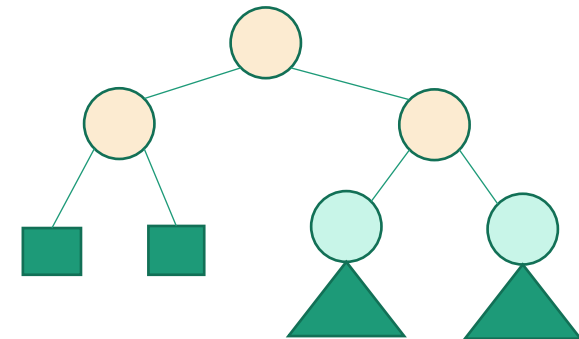
- Inductive step:

- **TO DO**



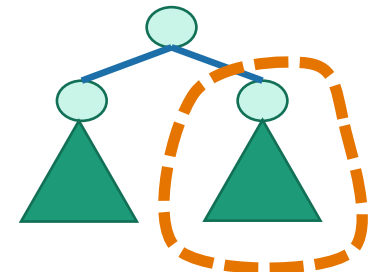
- Conclusion:

- after the last step,
 - there is an optimal tree containing this whole tree as a subtree.
- aka,
 - after the last step the tree we've constructed is optimal.



What have we learned?

- ASCII isn't an optimal way to encode English, since the distribution on letters isn't uniform.
- Huffman Coding is an optimal way!
- To come up with an optimal scheme for any language efficiently, we can use a **greedy algorithm**.
- To come up with a **greedy algorithm**:
 - Identify **optimal substructure**
 - Find a way to make “safe” choices that **won't rule out an optimal solution**.
 - Create subtrees out of the smallest two current subtrees.



Recap I

- Greedy algorithms!
- Three examples:
 - Activity Selection
 - Scheduling Jobs
 - Huffman Coding



Recap II



- Greedy algorithms!
- Often easy to write down
 - But may be hard to come up with and hard to justify
- The natural greedy algorithm may not always be correct
 - You'll see this on HW7
- A problem is a good candidate for a greedy algorithm if:
 - it has optimal substructure
 - that optimal substructure is **REALLY NICE**
 - solutions depend on just one other sub-problem.

Next time

- Greedy algorithms for Minimum Spanning Tree!